FlexFloat: A Software Library for Transprecision Computing

Giuseppe Tagliavini, *Member, IEEE,* Andrea Marongiu, *Member, IEEE,* and Luca Benini, *Fellow, IEEE*

***Abstract*—In recent years approximate computing has been extensively explored as a paradigm to design hardware and software solutions that save energy by trading off on the quality of the computed results. In applications that involve numerical computations with wide dynamic range, precision tuning of floating-point (FP) variables is a key knob to leverage the energy/quality trade-off of program results. This aspect assumes maximum relevance in the *transprecision computing* scenario, where accuracy of data is tuned at fine grain in application code. Performing precision tuning at fine grain requires a software development flow that streamlines the assessment of which variables have "precision slack" within an application. In this paper we introduce *FlexFloat*, an open-source software library that has been expressly designed to aid the development of transprecision applications. *FlexFloat* provides a C/C++ interface for supporting multiple FP formats. Unlike alternative libraries, *FlexFloat* enables to control the bit-width of mantissa and exponent fields and provides advanced features for the collection of runtime statistics, reducing the FP emulation time compared to the state-of-the-art solutions. Its design allows to emulate the behavior of standard IEEE FP types and custom extensions for reduced-precision computation. This makes the library suitable for adoption in multiple contexts, from manual exploration to integration into automatic tools. Experimental findings demonstrate that our approach can be used to perform a complete precision analysis from which deriving multiple program versions depending on the energy/quality trade-off. Furthermore, we show that the adoption of our methodology can lead to a significant reduction of energy consumption even on current commercial hardware (an embedded GPGPU).**

***Index Terms*—transprecision computing, floating-point emulation, precision tuning, energy-quality trade-off**

I. Introduction

The energy consumption of computing systems is constantly growing [1], which makes the development of energy-efficient design methodologies an evergreen research area. In this context approximate computing techniques [2] [3] have been proposed in a variety of domains to design hardware and software systems capable of trading *quality* of the computed results off for *energy* savings [4]. A wide variety of strategies has been explored in the literature, ranging from programming language approaches [5] [6] to transistor-level ones [7] [8] [9]. Among these strategies particular attention has been paid to *precision scaling* [10] [11] [12], a methodology which consists of changing the bit-width of program data to reduce storage and/or computing requirements.

Since most applications involving numerical computations with large dynamic range adopt floating-point (FP) data types, researchers have studied a number of software techniques focused on reducing the precision of FP formats [13] [14] [15] [16]. The execution of FP computations and related data transfers emerge as a major contributor to energy consumption; it has been shown that very significant power savings can be achieved by a careful combination of different precision levels for FP arithmetic [17] [18]. Nevertheless, in the general practice programmers usually assign the maximum precision provided by target platforms (32 bits, 64 bits or even more) to all program variables, following the most conservative approach to guarantee the precision of final results.

Recent research trends are moving towards a novel paradigm, known as *transprecision computing* [19] [20], in which rather than tolerating errors implied by imprecise hardware or software components systems are explicitly designed to systematically deliver "just enough quality". This is achieved by controlling approximation at a fine grain through the computation steps, not only focusing on final results but also tuning the precision of intermediate computations. In the context of FP workloads this is steering research efforts towards methodologies to understand which variables have "precision slack" within an application (i.e., variables for which precision requirements can be relaxed without impacting the results beyond the acceptable error threshold).

The IEEE Standard for Floating-Point Arithmetic (IEEE 754) [21] describes representation and memory encoding of five binary formats: *binary16* (half-precision), *binary32* (single-precision), *binary64* (double-precision), *binary128* (quadruple-precision) and *binary256* (octuple-precision). These formats defines the *dynamic range* as the ratio between the largest and smallest representable values, and the *precision* as the number of digits used to represent the mantissa. Considering for instance the C language, single-precision and double-precision formats correspond to `float` and `double` primitive types; on some platforms quad-precision is also available as `long double`, while octuple-precision is rarely implemented and smaller formats are typically not supported.

The adoption of smaller formats (16 bits or even less) offers significant potential to reduce the energy consumption of FP

This work has been partially supported by the European H2020 FET project OPRECOMP under Grant 732631.

G. Tagliavini and L. Benini are with the Department of Electrical, Electronic and Information Engineering "Guglielmo Marconi" (DEI), University of Bologna. e-mail: {giuseppe.tagliavini, luca.benini}@unibo.it.

A. Marongiu is with the Department of Computer Science and Engineering (DISI), University of Bologna, Italy. Email: a.marongiu@unibo.it

L. Benini is also with the Department of Information Technology and Electrical Engineering of the Swiss Federal Institute of Technology Zurich (ETH Zurich). e-mail: {luca.benini}@iis.ee.ethz.ch

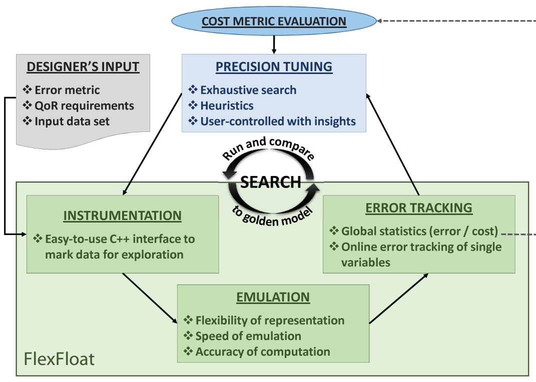
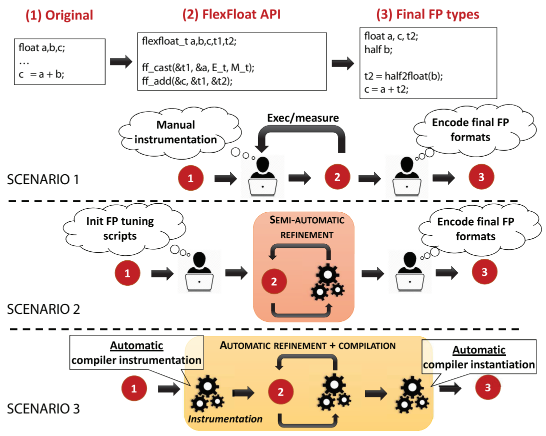


Fig. 1: FlexFloat: main uses and contributions.



```
float a,b,c;
…
c  = a + b;
```

```
flexfloat_t a,b,c,t1,t2;

ff_cast(&t1, &a, E_t, M_t);
ff_add(&c, &t1, &t2);
```

```
float a, c, t2;
half b;

t2 = half2float(b);
c = a + t2;
```

Fig. 2: Main application scenarios.

computations. We have compared the energy consumption of three versions of a linear algebra kernel (SAXPY) running on the NVidia Tegra X2 GPU (see Section **??** for details on our experimental setup), where we associate the program variables to FP types *binary16*, *binary32* and *binary64*, respectively. The adoption of *binary16* variables consumes 2.2× less energy than *binary32* variables, which in turn consume 1.9× less energy than *binary64*. This confirms that even in currently commercially available hardware reduced-precision variables bring quasi-linear energy savings (at the system level). However, the IEEE-standardized FP formats are only a small subset of the possible options in terms of total bit-width and of mantissa vs. exponent allocation of the available bits, and researchers have started to explore custom formats for future platforms.

Despite the major energy saving opportunity, system design and software development flows still lack a mature methodology to assess which FP variables in applications are amenable to precision reduction and which reduced-precision formats are most convenient. State-of-the art tools for precision tuning such as PROMISE [22] are often limited to standard data types (*binary32* and *binary64*), which misses the potential savings enabled by smaller formats (e.g., *binary16* is nowadays available on several platforms) and prevents their use for custom hardware design exploration. *fpPrecisionTuning* [15] relies on the GNU Multiple Precision Floating-Point Reliably (MPFR) [23] library to emulate arbitrary FP types, performing a heuristic search to find the minimum precision that can be associated to each variable in the program [24] [25] [26] [27].

While FP emulation coupled to precision tuning is the right approach to target both (i) precision reduction in applications being developed for existing hardware (targeting available IEEE FP formats) and (ii) the specification of custom FP formats for new hardware being designed, *state-of-the-art* FP emulation libraries such as MPFR [23] and SoftFloat [28] were not designed for this purpose and consequently have a number of shortcomings. Emulation libraries should be (i) **fast**, to minimize the time required to complete iterative precision tuning processes; (ii) **accurate**, to allow to correctly capture the accuracy that a custom HW type would deliver; (iii) **easy to augment**, as the definition of custom types should not require complex rewrites of the emulation library (the focus of application developers should stay on the application); (iv) **informative in terms of the achieved quality of results (QoR)**, as the convergence time of the search heuristics could be significantly reduced if better insight on the individual variables was propagated to the precision tuning tool.

In this paper we introduce ***FlexFloat***, an open-source software library[1] that has been expressly designed to aid the development of transprecision applications. *FlexFloat* defines a C/C++ Application Programming Interface (API) for supporting multiple (standard or custom) FP formats and arithmetic operations among them. Unlike other libraries, *FlexFloat*:

1) uses an emulation methodology that leverages native host platform types to significantly reduce the time required to emulate custom types. Our results show that *FlexFloat* is up to 2.8× and 2.4× faster than MPFR and SoftFloat, respectively;
2) enables accurate emulation of formats with arbitrary bit-width of mantissa and exponent fields (whereas MPFR does not allow to model the exponent field with a custom number of bits[2]);
3) requires no modification at all to support custom types (whereas the use of a new FP format in SoftFloat requires to deeply modify the library sources);
4) provides advanced statistics on program variables (casts and evolution of the error compared to a full-precision value) that can be used by optimization models to enhance the solution and/or decrease the search time (which no other emulation libraries provide).

Figure I highlights these concepts and visually shows what *FlexFloat* does and how it interacts with external tools. In this setting, the application designer takes care of defining a representative error metric and to outline QoR requirements for the application and input data set(s) at hand. Precision tuning tools first execute the program *as is* to derive a golden reference, then iteratively explore different precisions for FP variables, at each step comparing the result with the reference to guide the search heuristics.

[1] *FlexFloat* is available at this link: https://github.com/oprecomp/

[2] Only 32 or 64 bits words are used, which is adequate to precisely represent any value needed by scientific applications operating on very large numbers (the original application target of MPFR), but which can produce inaccurate results (and thus imply the derivation of wrong hardware specification) if used for hardware FP emulation, where a smaller number of bits is required.

Figure I depicts several conceptual scenarios *FlexFloat* can be used in. Programmers can **manually** (scenario 1) modify the program to adopt *FlexFloat*, then execute the program and compare the results with the golden model [29]. The process can be repeated multiple times by associating program variables to different precisions, with the aim to derive a final program version that uses the most convenient FP types. Programmers can **integrate the library with external tools for precision tuning** (scenario 2), making the exploration process (semi-)automatic. In this case the burden for the developer boils down to providing a wrapper interface for integration of the tools with the library. The error tracking at variable level can be used to guide the fine-grained tuning process. The third scenario is more forward looking, as it envisions the full **integration of *FlexFloat* into compilation toolchains**. This makes the FP precision tuning and the generation of the most efficient binary code for the target hardware/precision constraints (almost) entirely transparent to programmers.

Our experimental results section presents an assessment of the emulation performance of the library. We also report measurements for execution time and energy consumption on a commercial hardware platform executing different program versions (derived from the precision analysis), showing that *FlexFloat* leads to a significant reduction of energy consumption even on hardware that has not been fully designed based on transprecision principles. We conclude the experimental section with an example which illustrates the advanced features of the library.

The rest of the paper is organized as follows. Section **??** discusses the related work. Section **??** introduces the *FlexFloat* API and its common use cases. Section **??** describes the internals of the library. Section **??** introduces the framework setup and reports the experimental results. Finally Section **??** concludes and points out to future work directions.

II. Related work

The research area of approximate computing [2] [3] includes a wide set of techniques, ranging from software-oriented methodologies [5] [6] to ad-hoc hardware designs [7] [8] [9], with the aim to increase performance and power efficiency of computing systems. In this field many approaches deal with "statistical approximation", i.e., saving energy by running on unreliable but energy-efficient hardware. In the last years several tools have been released to perform statistical analysis and compute error bound of approximate software running on unreliable hardware [30] [31] [32]. In this paper we focus on approximation and energy reduction by executing programs on low-precision but deterministic (i.e., fully reliable) hardware, which is currently a major industry trend, especially thanks to the deep learning boom – as applications in this domain lend themselves to run extremely efficiently on low-precision hardware [33] [34].

Focusing on commercial hardware platforms with support for reduced-precision FP types, GPUs are widely used in several computing domains and reducing their energy consumption by adopting reduced-precision FP types is now widely acknowledged as a viable approach for several application domains. Mukunoki et al. [24] explore custom FP formats for GPUs. They show that performance can be significantly improved at whole-system level going to smaller formats and forcing data word-aligned memory locations; adopting these techniques the memory interface will use fewer transactions, which are among the most significant contributors to system energy consumption. NVidia Pascal has been the first GPU architecture to provide support to the *binary16* type, with major benefits for compute intensive domains such as machine learning [35] and linear algebra [36], and the new micro-architecture (Volta) further extends support to reduced precision types (e.g. mixed-precision multiply-and-add instructions). In this work we perform a set of experiments on this class of GPUs to show how *FlexFloat* can be practically used to aid the development of transprecision applications to reduce their energy consumption.

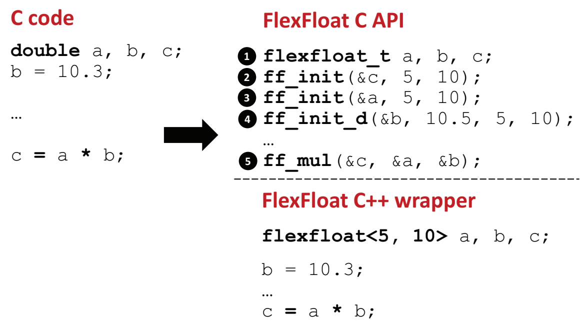
All commercial hardware platforms implement a subset of the IEEE 754 standard, whose limitations are starting to be highlighted by researchers. For example, the universal number format (*unum*) [37] has been recently proposed to overcome the problem of unsupervised accumulation of rounding errors. *unum* extends the IEEE standard by adding a metadata field (called *utag*) which enables support to variable-width storage and interval arithmetic [38]. The main shortcomings of *unum* – a very high hardware cost implied for practical implementations and a very high code refactoring effort – have generated some criticisms. A recent evolution of the specification, known as *unum-v3* or *posit* [39], mitigates both problems, which might in the future make its practical adoption feasible. Moreover, the IEEE FP formats represent only a small subset of all the possible options that custom hardware design could implement. In this field recent works have proposed dedicated FP units for reduced-precision [25] [26] and variable-precision [27] arithmetic. Focusing more specifically on reconfigurable hardware platforms (e.g., FPGAs), analytical techniques for range and precision analyses can be adopted to design fixed-point arithmetic circuits which guarantee the accuracy of hardware-implemented algorithms minimizing area and power consumption [40] [41]. As discussed in Section **??**, *FlexFloat* can be used inside compilation toolchains to perform precision tuning and produce efficient binary code tailored to the execution on any reduced-precision or tunable hardware unit.

In the area of FP emulation, several arithmetic libraries have been proposed to perform calculations on numbers with arbitrary precision. The GNU Multiple Precision Arithmetic Library (GMP) [42] has been one of the first libraries for arbitrary-precision arithmetic; it supports integers, rational numbers and also FP numbers. The GNU Multiple Precision Floating-Point Reliably (MPFR) [23] is based on GMP, adding to its arbitrary-precision representation the support for rounding modes, exceptions and special values as defined in the IEEE 754 standard. These libraries provide a basic C interface, but wrappers exist for other languages. For instance *Boost Multiprecision* [43] is a C++ library that can adopt both GMP and MPFR as a backend interface. ARPREC [44] is a library natively written in C++ with support for high-precision real, integer and complex types. These libraries are widely used in application areas where an "almost unbound" dynamic range is required and higher computation time is considered an unavoidable side-effect, such as some areas of scientific computing [45]. However they are not suitable for the exploration of reduced-precision FP types to increase performance and energy efficiency. As discussed in Section **??**, these libraries lack important features such as exponent bounding and the collection of execution statistics. Some features could be supported by refactoring the code at program level, for instance exponent bounding could be obtained by checking the exponent field after any operation, but this solution is neither general nor efficient in terms of emulation time. In Section **??** we show that *FlexFloat* provides up to 2.8× faster emulation than other libraries.

A different approach to emulate arbitrary FP types consists of emulating FP computations using a series of simpler fixed-point arithmetic operations that run on the integer arithmetic logic unit. *SoftFloat* [28] is a library that implements standard IEEE formats, enabling a bit-accurate emulation of the FP operations performed by FP hardware units. *SoftFloat* includes support for IEEE 754 types and can be extended to support custom formats. Some operations on custom formats are slower, since the library executes all the computations in software. This issue can be solved by using acceleration techniques based on SIMD integer units [46]. However the use of a new FP format requires to deeply modify the library sources, while *FlexFloat* allows to describe any format by invoking an API function to initialize the variable format.

Many research tools are available to perform automatic or semi-automatic precision tuning of program variables. *fp-PrecisionTuning* [15] implements a distributed algorithm to find the near-optimal precision for each FP variable in the program. Its main configuration parameter is the precision of the result, expressed as a value of signal-to-quantization-noise ratio (SQNR) that program results must satisfy. The tool executes the program multiple times, performing a heuristic search of the minimum precision that can be associated to each variable (for a fixed input set). A second phase performs a statistical refinement to join the precision bindings derived from different input sets. PROMISE [22], *Precimonious* [47] and *Blame analysis* [48] adopt more advanced techniques but their search space is restricted to a small subset of FP types (*binary32* and *binary64*).

All the discussed approaches for precision tuning adopt heuristic and simulation-based approaches. Recent efforts in rigorous FP error estimation are based on combinations of abstract interpretation and conservative range calculations. *Gappa* [49] and *Boost Interval library* [50] are tools based on interval arithmetic. *FPTuner* [16] is a rigorous tool for automatic precision-tuning of real-valued expressions, it generates a mixed-precision allocation (single, double, or quadruple precision) on a given input domain that is guaranteed to have error below a given threshold. PRECISA (Program Round-off Error Certifier via Static Analysis) [51] is a tool for the automatic estimation of round-off errors of functional expressions with an associated real domain (i.e., an interval). As a general consideration, these approaches are orthogonal and synergistic with our work; in addition most tools for precision tuning make use of MPFR or primitive types, but they can be easily extended to support *FlexFloat* with the aim to overcome MPFR limitations that have been highlighted earlier.



```
double a, b, c;
b = 10.3;

…

c = a * b;
```

```
❶ flexfloat_t a, b, c;
❷ ff_init(&c, 5, 10);
❸ ff_init(&a, 5, 10);
❹ ff_init_d(&b, 10.5, 5, 10);
…
❺ ff_mul(&c, &a, &b);
```

```
flexfloat<5, 10> a, b, c;
b = 10.3;
…
c = a * b;
```

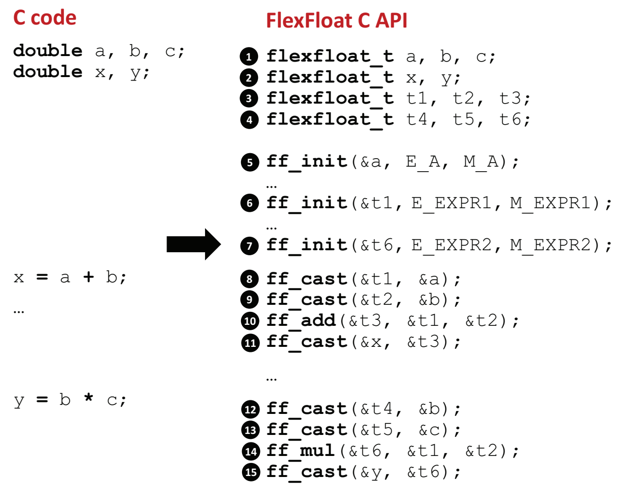
Fig. 3: C code transformed to use the *FlexFloat* API.

III. Using FlexFloat

A. Basic concepts

Figure III-A shows by means of an example how a C program has to be transformed to use *FlexFloat* primitives. First, the native FP types provided by the C language must be replaced with `flexfloat_t` types (referred to as *target type* in the following) (line 1). Before its first use each *FlexFloat* variable must be given an initial value for exponent and mantissa bit-widths (two unsigned integers). In lines 2-3 we invoke function `ff_init` to set 5 bits for the exponent and 10 bits for the mantissa, which characterize the IEEE 754 half-precision formatas the target type of the declared variables. Users can also (optionally) specify an initialization value expressed as a native C type (line 4). Since an initialization value might not be exactly representable in a target type with a lower number of bits, it is typically rounded to its nearest representable value (for advanced details on the rounding modes see Section **??**). Second, the *FlexFloat* API includes a set of functions to perform arithmetic operations involving operands of the same FP type. Using an approach common to other C libraries for arbitrary precision arithmetic (e.g., MPFR), such operations must be replaced by function calls that implement equivalent functionality on top of the emulated types (line 5).

FlexFloat also comes with a C++ wrapper, which further raises the level of abstraction for the replacement of FP types/operations in a program. The adoption of a C API combined with the availability of a C++ wrapper is a common solution for the development of software libraries whenever they are intended for multiple use scenarios. The C++ wrapper provides a generic FP type by defining a template class (`flexfloat<e,m>`) and a set of auxiliary functions for debugging and collecting statistics. This only requires users to replace original variable declarations with instantiations of this template class. No other part of the program needs modification since class methods include operator overloading (see Figure III-A). While the C++ wrapper simplifies the manual use of *FlexFloat* for FP precision tuning in target applications (less modifications requires, type checking upon



```
double a, b, c;                 1  flexfloat_t a, b, c;
double x, y;                    2  flexfloat_t x, y;
                                3  flexfloat_t t1, t2, t3;
                                4  flexfloat_t t4, t5, t6;

                                5  ff_init(&a, E_A, M_A);
                                   …
                                6  ff_init(&t1, E_EXPR1, M_EXPR1);
                                   …
                                7  ff_init(&t6, E_EXPR2, M_EXPR2);
x = a + b;                      8  ff_cast(&t1, &a);
…                               9  ff_cast(&t2, &b);
                                10 ff_add(&t3, &t1, &t2);
                                11 ff_cast(&x, &t3);
                                   …
y = b * c;                      12 ff_cast(&t4, &b);
                                13 ff_cast(&t5, &c);
                                14 ff_mul(&t6, &t1, &t2);
                                15 ff_cast(&y, &t6);
```

Fig. 4: A slightly more complex C code snippet with intermediate *FlexFloat* temporaries to improve precision tuning.

template instantiation with error reports in case of type mismatches, etc.), the lower-level C interface is better suited for integration with external tools, since it facilitates the creation of an optimized binary for which native bindings are provided by all the mainstream compilers (e.g., gcc and LLVM) and programming environments (e.g., Java and Python)[3].

B. Precision tuning

Figure III-B shows a slightly more complex C code where the same variables are used in multiple expressions. In the original code there are 5 variables, while the *FlexFloat* version declares 11 variables (lines 3-4). An additional variable is declared for each operand of an expression, with the aim to expose the precision of all intermediate computations during the tuning process. The `flexfloat_t` instances replacing the original variable declarations initialize their exponent and mantissa bit-widths with variable-specific values (i.e., `E_A` and `M_A` for variable `a` in line 5). The `flexfloat_t` instances representing temporary variables initialize their exponent and mantissa bit-widths with values that are specific of the expression they are invoked in (i.e., the precision of the target operator). In Figure III-B, `t1`, `t2` and `t3` are initialized with `E_EXPR1` and `M_EXPR1`, while `t4`, `t5` and `t6` are initialized with `E_EXPR2` and `M_EXPR2` (lines 6-7). Since all operations must be performed on operands of the same type (lines 10 and 14), operands are initially converted to the type of the related expression using the intermediate variables by invoking `ff_cast` (lines 8-9 and 12-13); finally the result is cast to its target type (lines 11 and 15).

All the values introduced in the program to represent exponents and mantissas (`E_i` and `M_i`) are defined as preprocessor symbols, so that a different version of the program can be obtained simply changing the value associated with these tunable parameters. Hence precision tuning is performed

[3]As the exploration space for non-trivial applications is typically huge, automated approaches will become mandatory for establishing solid transprecision computing methodologies. Much of our engineering effort has thus gone into optimizing the design of the low-level C interface.

```
half a, b;
float c;
half x;
float y;

x = a + b;
…
y = half_to_float(b) * c;
```

Fig. 5: Platform-specific code derived from precision tuning of the example in Figure III-B.

TABLE I: Result of precision tuning for the code in Figure III-B.

E_A	5 bits	M_A	10 bits
E_B	5 bits	M_B	10 bits
E_C	8 bits	M_C	23 bits
E_X	5 bits	M_X	10 bits
E_Y	8 bits	M_Y	23 bits
E_EXPR1	5 bits	M_EXPR1	10 bits
E_EXPR2	8 bits	M_EXPR2	23 bits

assigning different values to `E_i` and `M_i` parameters and performing an evaluation of the result quality for each version that has been explored. In our example, suppose that `x` and `y` are the program outputs, with precision requirements defined as $|x - x_{ref}| \leq 10^{-4}$ and $|y - y_{ref}| \leq 10^{-6}$ (x_{ref} and y_{ref} are the output values of the original program). The exploration can be performed by substituting `E_i` and `M_i` with the values corresponding to the FP types supported by the target platform. For each assignment of all parameters the program must be executed to check the result quality against requirements.

Considering a concrete instance of the example depicted in Figure III-B, we perform precision tuning for a target architecture including three standard FP formats (*binary16*, *binary32* and *binary64*). Table III-B shows the final outcomes of the tuning process. Using these numbers, programmers can derive a version using the primitive FP types available on the target platform (Figure III-B). `a`, `b` and `x` can be declared as *binary16* variables, `x` and `y` as *binary32* variables. Since the type of the second expression is *binary32* while the type of `b` is *binary16*, a cast is required.

C. Execution statistics

The *FlexFloat* API includes functions to start, stop and reset the collection of *execution statistics*. The final report includes the number of arithmetic operations (grouped by operator name) and the number of casts (grouped by source+destination type pairs). This is a key feature of *FlexFloat*, since it allows to evaluate the overhead due to the *casts* that have been introduced by the transprecision computing transformation methodology, where the precision of computations is changed at a fine granularity.

These statistics can be used to derive policies that drive the transformation of a program into a new mixed-precision version by taking into account the overhead introduced by casts (see Section **??**). This feature is totally transparent to the user and does not impact the usability of *FlexFloat*.

```
1  flexfloat_t sum, data[N];
   ...
2  ff_init_d(&sum, 0.0, 5, 10);
3  ff_track_callback(&sum, plot, &i);
   ...
4  for(i=0; i<N; i++) {
5    ff_acc(&sum, data[i]); // sum += data[i];
   }

 void plot (flexfloat_t *v,
            void *arg) {
   ...
6   i = *((int *)arg);
7   v = ff_track_get_exact(v);
8   e = ff_track_get_error(v);
9   rel = e * 100 / v:
10  add_to_chart(i, rel);
 }
```

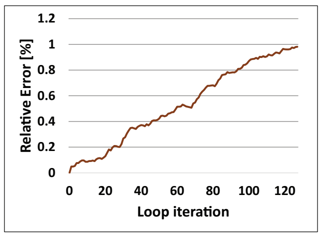


Fig. 6: Example of fine-grain tracking: Evolution of the relative error for a single variable during program execution.

FlexFloat provides an advanced feature to track at fine grain the accumulation of errors on program variables. When this feature is enabled, programmers (or automatic tools) can retrieve the exact value of a computation stored in a variable (or its current error w.r.t. the exact value) at any point in the program. In addition users can add a callback to a program variable, that is a function invoked at any update of the variable. This feature can be useful for different purposes, for instance it can be used to track which internal expression has more impact on the result quality, to study the evolution of the error over time to implement custom cost metric evaluation (e.g., the cost of cast operations) to be used to steer the precision tuning heuristics.

Figure III-C shows an example of fine-grained tracking of the result of an expression inside a loop (lines 4–5). A callback function is set for variable `sum` in line 3; `plot` is invoked at each update of the related variable (the accumulation in line 5) with two parameters. The first parameter of a callback function is always the modified variable while the second parameter is custom and its purpose is context-dependent. In this example the second parameter is the loop variable `i`, which is converted back to an integer variable (line 6). Calls to `ff_track_get_exact` (line 7) and `ff_track_get_error` (line 8) are used to retrieve the values required to compute the relative error (line 9). Finally, this value is used to build a x-y plot of the error evolution over multiple loop iterations (as shown in Figure III-C). This feature enables the adoption of sample-based methodologies for error analysis, which can be more focused on critical variables compared to end-to-end metrics.

IV. FLEXFLOAT INTERNALS

The following sections provide more insight on the internals and the design choices of *FlexFloat*.

A. Internal representation of FP types

FlexFloat provides an ADT (`flexfloat_t`) which encapsulates two unsigned integers (e and m) and a FP field (v). e and m represent the number of bits used for exponent and mantissa. Fixed values for e and m determine unambiguously a target type. v contains the current FP value for an instance of the target type and it is referred as *backend value*. The type of v is fixed for all instances of the *FlexFloat* ADT, it can be chosen at compile time among `float`, `double` (default) and `_Float128`[4]. In the rest of the paper we will consider a backend value of `double` type for practical reasons, since it is the highest precision considered in our benchmark suite; however the adoption of `_Float128` as backend value enables the emulation of FP types up to quad-precision.

The format of a generic target type follows the conventions of IEEE standard:

- the binary representation includes 1 bit for the sign, e bits for the exponent and m bits for the mantissa: $b_{m+e}b_{m+e-1}...b_m b_{m-1}...b_0$;
- the value encoded in the exponent field is biased by $2^{e-1}-1$;
- the associated real value is: $-1^{b_{m+e}} \times 0.(b_{m-1}...b_0)_2 \times 2^{(b_{m+e-1}...b_m)_2-bias}$
- the values 00..0 and 11..1 of the exponent are dedicated to encode the special cases considered by the standard, that are ± 0, denormal numbers, $\pm$infinity and not-a-number (NaN).

Figure IV-A depicts the format of a generic FP type, reporting the number of bits dedicated to exponent and mantissa in IEEE 754 formats.

When a primitive value is provided at initialization time, it is stored into the backend value and then it is *sanitized*. More in detail, the sanitizing process for a FP type performs the following steps to derive the backend value:

- a primitive value equal to $\pm$infinity or NaN is encoded as-is in `double` format;
- a primitive value that exceeds the range of the target type is encoded as $\pm$infinity in `double` format;
- a primitive value that is smaller than $\pm 2^{-2^{n-1}-2}$ is converted to a denormalized notation; if its exponent is smaller than $-2^{n-1}-2$ then it is encoded as ± 0;
- in all other cases the result is encoded in `double` format, i.e. the sign bit is preserved, the exponent is biased by 1023 and the $53-e$ least significant bits of the mantissa are set to zero.

After these steps, the backend value contains a value encoded in `double` format which conveys the same precision and dynamic range of the correspondent value encoded in the target format. The sanitizing step is always transparent to *FlexFloat* users.

B. Arithmetic operations and casts

Arithmetic operations are directly performed on backend values in `double` format. An arithmetic function initializes a `flexfloat_t` instance of the same type of operands width and store the result in the backend value, which is sanitized and returned as result. This methodology guarantees shorter

[4]This format is introduced by ISO/IEC TS 18661-3:2015 standard. Note that on most architectures `long double` is not mapped on a quad-precision FP type.

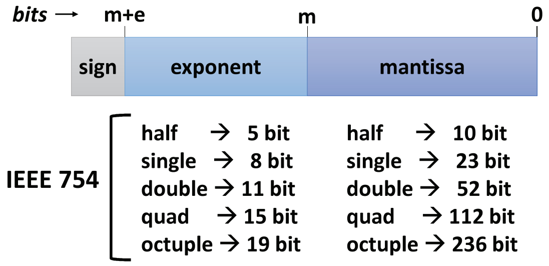


Fig. 7: Format of a generic FP type and number of bits dedicated to exponent and mantissa in IEEE 754 formats.

execution times w.r.t. an approach based on full emulation of arithmetic hardware units (e.g., *SoftFloat*) while preserving bit-level accuracy.

FlexFloat provides a specific API function to perform a conversion between different FP types. In detail, a cast from a variable a of type A to a variable b of type B is performed by sanitizing the backend value of a with the sanitize function of B and finally storing the result in the backend value of b.

C. Rounding modes

FlexFloat supports the four rounding rules defined in the IEEE 754 standard, that are:

- *round-to-nearest* (default) – results are rounded to the nearest representable value;
- *round-to-0* – results are rounded to the largest representable value whose magnitude is less than that of the result (truncation);
- *round-toward-*$+\infty$ – results are rounded to the smallest representable value which is greater than the result (ceiling);
- *round-toward-*$-\infty$ – results are rounded to the largest representable value which is less than the result (floor).

The current rounding mode is set by calling the `fesetround` function declared in the `<fenv.h>` header file, which is defined by the ISO C standard for FP computations. In *FlexFloat* the rounding process is realized as a preamble to the sanitize step, applying an adjustment to the backend value whenever it is required to produce a correct result. Considering for instance the *round-to-nearest* mode, an addendum of $-1^{b_{m+e}} \times 2^{53-m}$ is summed to the backend value if and only if the mantissa bit at position $53 - m - 1$ is equal to 1. This design allows users to selectively disable the support to rounding modes to speed up program execution. *round-to-0* mode is automatically applied when rounding support is disabled with the aim to minimize the library overhead (see the experimental results in Section **??**).

V. Experimental Evaluation

We compare *FlexFloat* with other libraries in terms of emulation time (Sec. **??**), we show how it can be used to perform precision tuning on existing GPU hardware for improved performance and energy (Sec. **??**) and we discuss the use of its advanced features (Sec. **??**).

We have selected a set of benchmarks implemented in C which use the `double` type for all FP variables. Our benchmark suite includes ten programs from different application domains. Linear algebra kernels (SAXPY, FWT) are used by many applications, recent surveys report that almost 50% of these kernels are amenable to lower-than-32-bits precision calculations [52]. Several algorithms for image processing make use of prefix sum (SCAN) as a preliminary step; its reduced-precision versions are typically used in resource-constrained embedded systems [53]. In the field of computational finance (BSCHOLES) a research study shows that adopting lower precision types in Black-Scholes code provides more precise outputs than other algorithms (e.g., Monte Carlo simulation) [54]. The area of scientific computing (JACOBI) is one of the most reluctant at introducing reduced-precision computations, but some recent works follow this direction with the aim to better exploit the computational power of GPUs [55] [56] [57]. Reduced-precision convolutions (CONV) are widely used by machine learning algorithms to improve the accuracy of both training and inference of deep neural networks [58] [33]. Fixed-point versions of discrete wavelet transforms (DWT) and support vector machines (SVM) have been demonstrated to be beneficial for energy efficiency in designing seizure detection algorithms for resource-constrained embedded systems [59]. Their analysis can be naturally extended to reduced-precision FP types. Algorithms for data mining (KMEANS, CORR) can use half-precision arithmetic to perform fast classifications and similarity search on GPUs [60] [61].

We target the embedded GPU on the NVidia Drive CX2 development board; this platform includes a Tegra X2 SoC which contains 2 Denver cores, 4 ARM A57 cores and a GPU from the Pascal generation with 256 CUDA cores grouped in two Streaming Multi-processors (SMs) sharing a 512KB L2 (last-level) cache. This platform supports three standard FP data types (*binary16*, *binary32*, *binary64*) and provides efficient conversion operations in its instruction set which can be used to scale precision at runtime. Moreover, the compute pipeline includes 2-way vector half-precision arithmetic units which can issue two *binary16* operations at the same rate as a single *binary32* operation. Overall half-precision arithmetic has twice the throughput of single-precision arithmetic and four times the throughput of double precision.

In our experiments on precision tuning we have considered as a quality metric the value of signal-to-quantization-noise ratio (SQNR) computed over result elements in the benchmark result set. Concerning the quality of results, we consider four different requirements expressed as four SQNR values ($10^{10}, 10^5, 10^3, 10$: the higher the value the smaller the tolerated error).

To evaluate the energy consumption of the target platform we use a digital multimeter to measure the current consumption of the entire board with only an Ethernet connection and no other off-board peripherals. Combining the current measurement with the input voltage of the board (12 V) and the execution times of a native CUDA version of the program (discussed in Section **??**) we estimate the energy consumption of each benchmark.

TABLE II: Percentage of variables associated to a FP type (*binary16*, *binary32*, *binary64*) by the precision tuning process, considering four alternative targets of result quality.

	SQNR											
	10^{10}			10^5			10^3			10		
Type / Bench.	binary64	binary32	binary16	binary64	binary32	binary16	binary64	binary32	binary16	binary64	binary32	binary16
SAXPY	0%	66.7%	33.3%	0%	0%	100%	0%	0%	100%	0%	0%	100%
SCAN	0%	100%	0%	0%	66.7%	33.3%	0%	0%	100%	0%	0%	100%
BSCHOLES	100%	0%	0%	0%	73.3%	26.7%	0%	40%	60%	0%	26.7%	73.3%
JACOBI	100%	0%	0%	0%	48%	52%	0%	48%	52%	0%	36%	64%
CONV	0%	100%	0%	0%	25%	75%	0%	0%	100%	0%	0%	100%
SVM	0%	100%	0%	0%	83.3%	16.7%	0%	33.3%	66.7%	0%	33.3%	66.7%
DWT	100%	0%	0%	0%	71.4%	28.6%	0%	71.4%	28.6%	0%	71.4%	28.6%
KMEANS	0%	100%	0%	0%	80%	20%	0%	0%	100%	0%	0%	100%
FWT	100%	0%	0%	0%	100%	0%	0%	100%	0%	0%	100%	0%
CORR	0%	42.9%	57.1%	0%	42.9%	57.1%	0%	42.9%	57.1%	0%	42.9%	57.1%

A. Emulation results

We have performed a set of experiments to compare the emulation time of the current implementation of *FlexFloat* with two libraries, *SoftFloat* (version 3d) and MPFR (version 3.1.3). We have designed a synthetic benchmark which includes an instance of all arithmetic operators with forward data dependencies among variables (i.e., the result of an expression is used as operand in the next one); operands are initialized with random values and operations are executed in a single loop of one billion iterations. We have implemented three variants for FP emulation, one for each API. Since *SoftFloat* does not support arbitrary types we have focused our analysis on *binary16*; similar considerations are valid for the emulation of any *smaller-than-32-bits* type. In the MPFR version we have set the precision parameter to size of the mantissa of a *binary16* type (11 bits), but it is not possible to limit the exponent to the 5 bits mandated by the IEEE standard. To prevent unfair comparisons with those APIs that limit the exponent to 5 bits the benchmark has been designed to prevent overflows in the dynamic range. The source code has been compiled using GCC 4.9 with optimization level O3 and target x86_64-linux-gnu, then executed on a Core i7-6500U CPU.

Figure V-A reports the execution times of the different versions of the synthetic benchmark compared to the baseline *FlexFloat (fast math)*; in this configuration both rounding support and execution statistics are disabled with the aim to optimize arithmetic computations to the detriment of strict IEEE compliance. The *Native* case reports the execution time when using the native `double` type for all variables: This case is clearly much faster than any emulation scheme, as the `double` type is natively supported in hardware. The numbers labeling the arrows report the slow-down of the emulation time compared to the baseline. *MPFR* is 2.8 times slower than *FlexFloat (fast math)*, while *SoftFloat* is 2.4 times slower. When the emulation of the various rounding modes is enabled the emulation speed of *FlexFloat* is obviously reduced: Even in this case it is 31% and 11% faster than MPFR and *SoftFloat*, respectively. In case both rounding modes and execution statistics are activated the execution time of *FlexFloat* is equivalent to the other libraries; however in this configuration the library is providing unique advanced features and consequently these results cannot be directly compared.

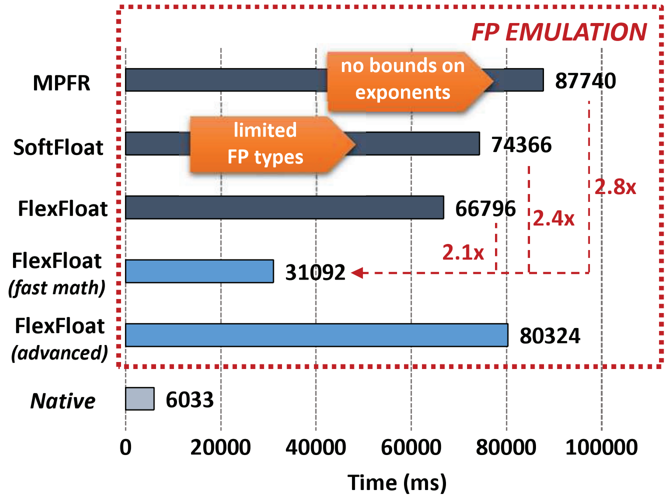


Fig. 8: Comparison with MPFR and SoftFloat.

Precision tuning. Next, we couple *FlexFloat* to a precision tuning tool [15]. The tuning process analyzes multiple configurations in which each FP variable is assigned to one of the available types. The tuner re-executes the program for each configuration and computes the error on its output values to provide a measure of the result accuracy[5].

Table V-A reports the percentage of variables that are associated to different FP types for each benchmark at the end of the tuning process. If no relaxation on the output quality is considered (SQNR = 10^{10}) a large number of program variables will be mapped onto high precision types (*binary64* and *binary32*). As the error tolerance increases (SQNR gets lower) a higher number of program variables is mapped onto low precision types.

In some (lucky) cases all the variables can be lowered in precision to the same FP type (e.g., all variables in SAXPY are half- precision for any value of SQNR less than 10^10), but in general this is not true. When variables cannot be all lowered to the same type cast operations are required. As such operations are costly, it is important to assess the trade-off be-

[5]Different error metrics can be provided by passing a function as a parameter to the script.

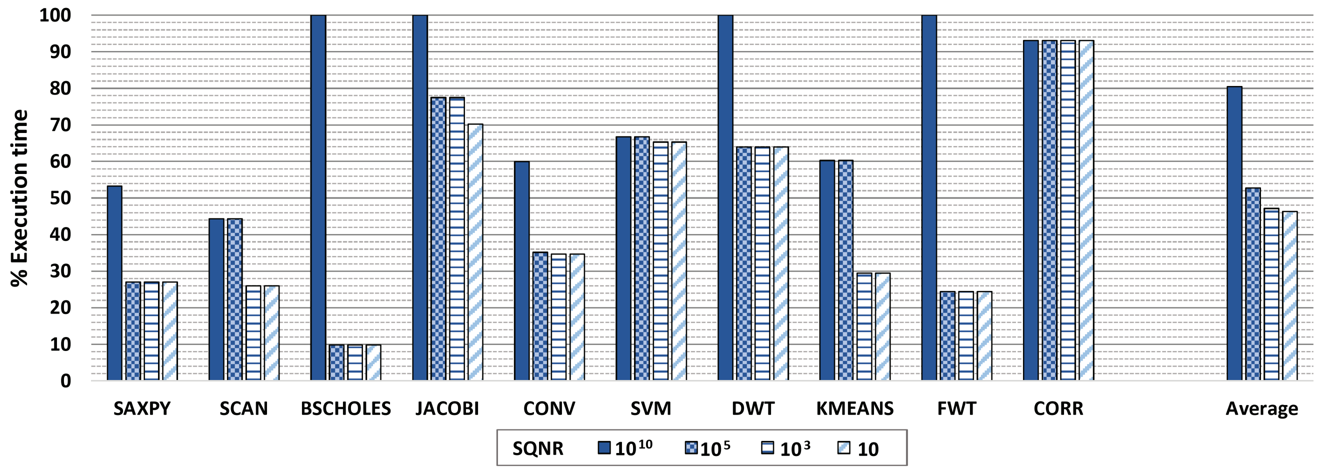


Fig. 9: Execution time of each benchmark normalized to the *double* version.

TABLE III: Ratio between the number of casts and the number of FP operations after precision tuning.

SQNR / Bench.	10^{10}	10^{5}	10^{3}	10
SAXPY	0.50			
SCAN		1.00		
BSCHOLES		0.16	0.10	0.14
JACOBI		0.60	0.30	0.30
CONV		0.35		
SVM		0.54	0.10	0.10
DWT		0.26	0.26	0.26
KMEANS		0.60		
FWT				
CORR	0.21	0.21	0.21	0.21

tween this cost and the benefits introduced by lower precision arithmetic. Using *FlexFloat* we can obtain execution statistics – that no other library provides – which include the number of cast operations required at runtime. Table V-A reports the ratio between the number of casts and the number of FP operations (*Casts/Ops*) for all the configurations (benchmark and target quality) which require different FP types from the previous analysis. Casts add an overhead to the execution time that reduces the speed-up of the tuned version of a benchmark compared to the double-precision reference. We have empirically determined that mixed-precision computation is profitable when the value of *Casts/Ops* is smaller than 0.50 (i.e., the number of casts should be smaller than half the number of arithmetic operations).

B. Case study: FlexFloat *for transprecise GPU computation*

We have implemented multiple CUDA versions of each benchmark. The baseline version uses only double-precision variables (*binary64*), whereas the other versions use the mix of FP types shown in Table V-A. The mixed-precision versions always make use of vectorial FP types and operations in order to maximize the utilization of hardware units and the reduction in energy consumption. Figure V-A reports the execution time of each benchmark normalized to the baseline.

On average mixed-precision computation allows 20% (for the lowest accuracy relaxation) to 52% (for the highest accuracy relaxation) reduction in execution time. Looking at Table V-A we can see that benchmarks SAXPY, SCAN, CONV, KMEANS all achieve precision lowering to *binary16* for 100% of their program variables when SQNR = $\{10, 10^3\}$. This theoretically allows to reduce their execution time to 25% of the baseline (*binary64*). This is indeed the case for SAXPY and SCAN. The slightly higher execution time for CONV and KMEANS is justified by the fact that only part of the half-precision computations are vectorized due to the presence of control flow statements.

Although the totality of the program variables in CORR can be lowered in precision to a mix of *binary32* and *binary16*, this only enables a small reduction in execution time ($< 10\%$). The reason for this behavior is that a large portion of the program is spent over synchronization (CUDA barriers).

BSCHOLES shows higher-than-ideal reduction in execution time for SQNR = $\{10, 10^3, 10^5\}$. The baseline version of this benchmark is characterized by a high communication/computation ratio that forces long idle periods on the processing cores. The memory bandwidth request reduction implied by the adoption of the half-precision types has the side effect of significantly reducing the stall time.

Figure V-B depicts the energy consumption of each benchmark normalized to the double-precision version. On average the results follow the trends observed for the execution time plot, with an even more pronounced advantage in energy savings, which range from 22% (for the lowest accuracy relaxation) to 60% (for the highest accuracy relaxation). BSCHOLES and CORR deserve further discussion. The former seemingly exhibits lower benefits compared to the execution time results. This is due to the fact that while memory transfers and computations are overlapped in time, they are cumulative to the total energy consumption (note that energy consumption of the processing elements in idle/stall state is negligible). The latter shows higher improvements in energy efficiency compared to the execution time alone. As already mentioned, energy consumption in idle state is nearly negligible, which makes the large part (in execution time) of synchronizations irrelevant in terms of energy. As a consequence, the savings implied by the adoption of lower-precision arithmetic are

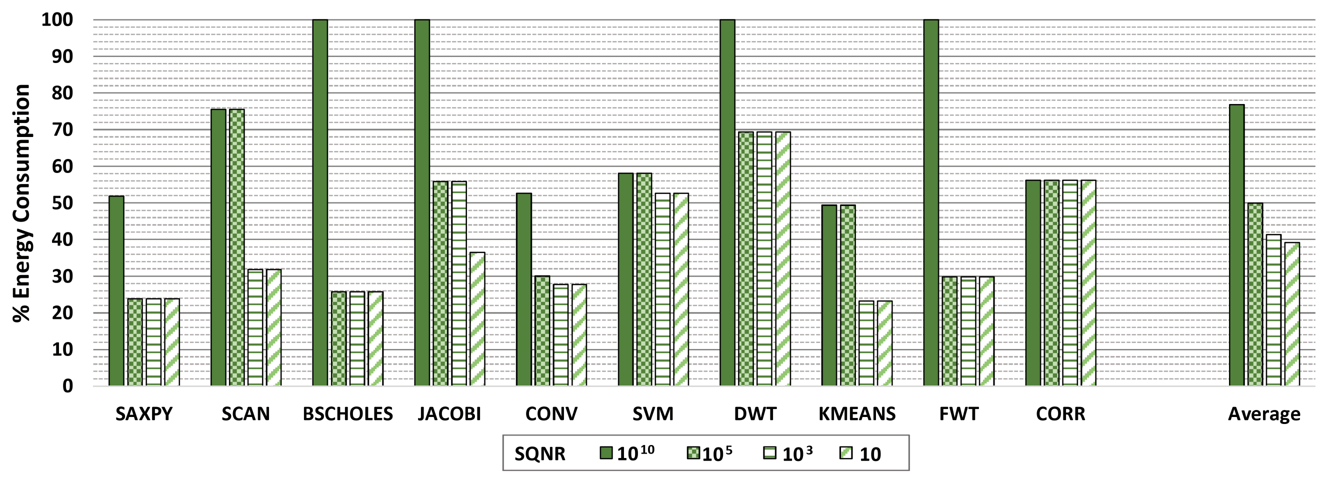
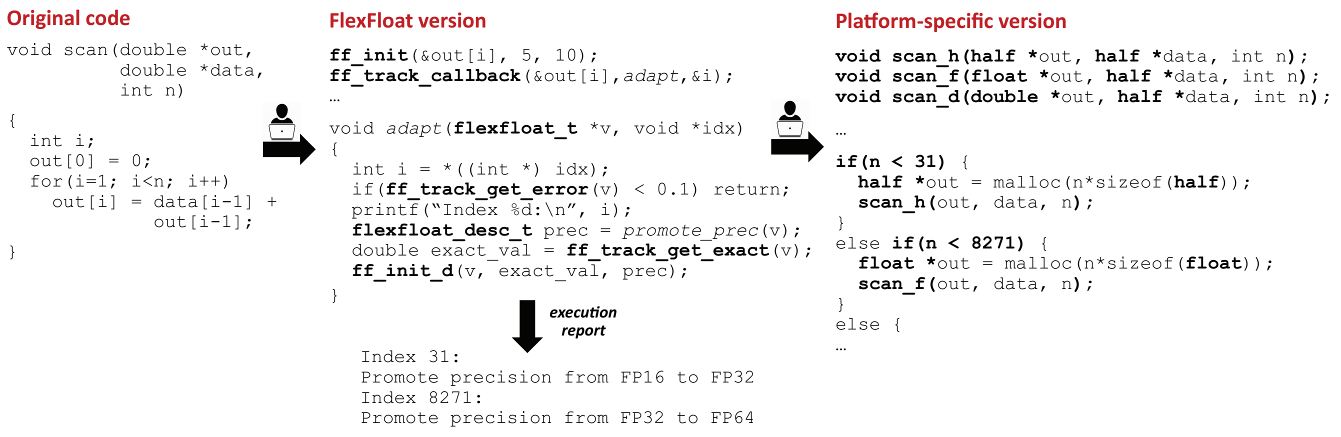


Fig. 10: Energy consumption of each benchmark normalized to the *double* version.



```
void scan(double *out,
          double *data,
          int n)
{
  int i;
  out[0] = 0;
  for(i=1; i<n; i++)
    out[i] = data[i-1] +
             out[i-1];
}
```

```
ff_init(&out[i], 5, 10);
ff_track_callback(&out[i],adapt,&i);
…
void adapt(flexfloat_t *v, void *idx)
{
  int i = *((int *) idx);
  if(ff_track_get_error(v) < 0.1) return;
  printf("Index %d:\n", i);
  flexfloat_desc_t prec = promote_prec(v);
  double exact_val = ff_track_get_exact(v);
  ff_init_d(v, exact_val, prec);
}
```

```
Index 31:
Promote precision from FP16 to FP32
Index 8271:
Promote precision from FP32 to FP64
```

```
void scan_h(half *out, half *data, int n);
void scan_f(float *out, half *data, int n);
void scan_d(double *out, half *data, int n);
…
if(n < 31) {
  half *out = malloc(n*sizeof(half));
  scan_h(out, data, n);
}
else if(n < 8271) {
  float *out = malloc(n*sizeof(float));
  scan_f(out, data, n);
}
else {
…
```

Fig. 11: Example of adaptive code based on a one-dimensional version of SCAN.

"magnified" in this plot.

It is worth noting that the energy savings for SCAN in the configurations SQNR = $\{10^5, 10^{10}\}$ are also smaller than the "savings" in execution time. This is a visible effect of the cost introduced by the cast operations. It is possible to see from Table V-A that the *Casts/Ops* value for these configurations is very high. This is an important observation, since it demonstrates the importance of integrating the cost of cast operations into optimization models for precision tuning. Overall, these results show that the adoption of *FlexFloat* is effective in exploring the trade-offs between quality and energy consumption in transprecision computing.

C. Advanced features

In this last part of the experimental section we show how the advanced profiling features of *FlexFloat* (introduced in Section **??**) can be used to identify parameters that are more critical for error propagation. Indeed the findings of this analysis can be used to derive multiple variants for limited portions of the original program, each one characterized by a specific precision binding for inner variables. Using this approach, a programmer can derive an *adaptive version* of the program that includes multiple code variants and the logic required to switch among them.

Figure V-B shows an example of adaptive code based on a one-dimensional version of SCAN. The quality requirement on the program output is $|out[i] - out_{ref}[i]| < 0.1$. A first analysis on a half-precision version of the program has confirmed the intuition that the computation of `out[i]` is the most critical statement for error accumulation, while the `data` array can be declared as half-precision since input data are provided in decimal format with only 3 significant digits. Moreover, the error on `out[i]` is heavily affected by the parameter `n`, that is the number of elements in the input array. We have explored the dependence between `n` and error propagation providing a new program version which uses the tracking feature of *FlexFloat*. More in detail, we have defined a callback function (`adapt`) that executes these steps:

1) if the current error is less than 0.1, it returns with no side effect;
2) (else) it calls the `promote_prec` function to promote the precision of the current output element to the next available one (from half-precision to single-precision and finally to double-precision);
3) it retrieves the exact value of the current output element;
4) it re-initializes the current output element with its exact value (from step 3) and the precision computed at step 2.

The execution of this code produces a report (Figure V-B, on the bottom) that highlights which iterations require a precision change to fulfill the quality constraint. We have used this report to derive a platform-specific version of the original program (Figure V-B, on the right), declaring three variants of the `scan` function. The logic required to switch among these variants checks the current value of the iteration variable.

VI. Conclusion

FlexFloat is an open-source software library designed to aid the development of transprecision systems by providing a C/C++ API for supporting multiple FP formats. *FlexFloat* enables the control of the bit-width of mantissa and exponent fields and provides advanced features for the collection of runtime statistics. These features allow to emulate the behavior of standard or custom FP types and make the library suitable for adoption in multiple contexts, from manual explorations to the integration with advanced tools for precision tuning. Experimental results show that *FlexFloat* outperforms other libraries in emulation speed, with up to 2.8× faster program execution. This is key to enable the exploration of multiple precision configurations within reasonable time. Moreover, the adoption of a methodology based on *FlexFloat* is effective in finding the best trade-off between quality and energy consumption on a real computing platform. On a commercial embedded GPU based on the NVidia Tegra X2 SoC we have observed that mixed-precision computation allows on average 20% to 52% reduction in execution time and 22% to 60% reduction in energy consumption for a wide set of benchmarks.

Our future work will be focused on integrating *FlexFloat* within toolchains aimed at automating the development of transprecision applications, enabling a conversion from source code to a *FlexFloat* representation that can be used to perform precision tuning. The results of this analysis can be injected in the compilation toolchain with a feedback loop and used to produce a final version based on primitive types available in the compiler backend. We also plan to extend the library to support more FP formats (e.g., *unum* and *posit*).

References

[1] M. Avgerinou, P. Bertoldi, and L. Castellazzi, "Trends in Data Centre Energy Consumption under the European Code of Conduct for Data Centre Energy Efficiency," *Energies*, vol. 10, no. 10, p. 1470, 2017.

[2] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate computing: A survey," *IEEE Design & Test*, vol. 33, no. 1, pp. 8–22, 2016.

[3] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys (CSUR)*, vol. 48, no. 4, pp. 1–33, 2016.

[4] M. Alioto, "Energy-quality scalable adaptive VLSI circuits and systems beyond approximate computing," in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2017, pp. 127–132.

[5] 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*. ACM, 2011, pp. 124–134.

[6] B. Nongpoh, R. Ray, S. Dutta, and A. Banerjee, "AutoSense: A Framework for Automated Sensitivity Analysis of Program Data," *IEEE Trans. on Software Engineering*, vol. 43, no. 12, pp. 1110–1124, 2017.

[7] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy, "Low-power digital signal processing using approximate adders," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 1, pp. 124–137, 2013.

[8] B. Li, P. Gu, Y. Shan, Y. Wang, Y. Chen, and H. Yang, "RRAM-based analog approximate computing," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 12, pp. 1905–1917, 2015.

[9] M. Imani, D. Peroni, and T. Rosing, "CFPU: Configurable Floating Point Multiplier for Energy-Efficient Computing," in *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 2017, p. 76.

[10] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate data types for safe and general low-power computation," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 164–174.

[11] C.-C. Hsiao, S.-L. Chu, and C.-Y. Chen, "Energy-aware hybrid precision selection framework for mobile GPUs," *Computers & Graphics*, vol. 37, no. 5, pp. 431–444, 2013.

[12] Y. Tian, Q. Zhang, T. Wang, F. Yuan, and Q. Xu, "Approxma: Approximate memory access for dynamic precision scaling," in *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*. ACM, 2015, pp. 337–342.

[13] C. Bekas, A. Curioni, and I. Fedulova, "Low-cost data uncertainty quantification," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 8, pp. 908–920, 2012.

[14] P. Klavík, A. C. I. Malossi, C. Bekas, and A. Curioni, "Changing computing paradigms towards power efficiency," *Philosophical Trans. of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 372, no. 2018, 2014.

[15] N.-M. Ho, E. Manogaran, W.-F. Wong, and A. Anoosheh, "Efficient floating point precision tuning for approximate computing," in *22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2017, pp. 63–68.

[16] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić, "Rigorous floating-point mixed-precision tuning," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, 2017, pp. 300–315.

[17] C. Bekas and A. Curioni, "A new energy aware performance metric," *Computer Science-Research and Development*, vol. 25, no. 3, pp. 187–195, 2010.

[18] S. Mach, D. Rossi, G. Tagliavini, A. Marongiu, and L. Benini, "A Transprecision Floating-Point Architecture for Energy-Efficient Embedded Computing," in *Circuits and Systems (ISCAS), 2018 IEEE International Symposium on*. IEEE, 2018, pp. 1–5.

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

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

[21] D. Zuras, M. Cowlishaw, A. Aiken, M. Applegate, D. Bailey, S. Bass, D. Bhandarkar, M. Bhat, D. Bindel, S. Boldo *et al.*, "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.

[22] S. Graillat, F. Jézéquel, R. Picot, F. Févotte, and B. Lathuilière, "Auto-tuning for floating-point precision with Discrete Stochastic Arithmetic," Jun. 2016, preprint. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01331917

[23] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Trans. on Mathematical Software (TOMS)*, vol. 33, no. 2, p. 13, 2007.

[24] D. Mukunoki and T. Imamura, "Reduced-Precision Floating-Point Formats on GPUs for High Performance and Energy Efficient Computation," in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2016, pp. 144–145.

[25] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar, "Reducing power by optimizing the necessary precision/range of floating-point arithmetic," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 8, no. 3, pp. 273–286, 2000.

[26] T. Rzayev, S. Moradi, D. H. Albonesi, and R. Manchar, "DeepRecon: Dynamically reconfigurable architecture for accelerating deep neural networks," in *International Joint Conference on Neural Networks (IJCNN)*, 2017, pp. 116–124.

[27] H. Kaul, M. Anders, S. Mathew, S. Hsu, A. Agarwal, F. Sheikh, R. Krishnamurthy, and S. Borkar, "A 1.45GHz 52-to-162GFLOPS/W variable-precision floating-point fused multiply-add unit with certainty tracking in 32nm CMOS," in *IEEE International Solid-State Circuits Conference*, 2012, pp. 182–184.

[28] J. R. Hauser, "Handling floating-point exceptions in numeric programs," *ACM Trans. on Programming Languages and Systems (TOPLAS)*, vol. 18, no. 2, pp. 139–174, 1996.

[29] A. Willème, A. Descampe, S. Lugan, and B. Macq, "Quality and Error Robustness Assessment of Low-Latency Lightweight Intra-Frame Codecs for Screen Content Compression," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 6, no. 4, pp. 471–483, 2016.

[30] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: Reliability-and accuracy-aware optimization of approximate computational kernels," in *ACM SIGPLAN Notices*, vol. 49, no. 10. ACM, 2014, pp. 309–328.

[31] B. Boston, A. Sampson, D. Grossman, and L. Ceze, "Probability type inference for flexible approximate programming," in *ACM SIGPLAN Notices*, vol. 50, no. 10. ACM, 2015, pp. 470–487.

[32] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve, "Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–14.

[33] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaev, G. Venkatesh *et al.*, "Mixed Precision Training," *arXiv preprint arXiv:1710.03740*, 2017.

[34] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-Eye: A Complete Design Flow for Mapping CNN Onto Embedded FPGA," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 35–47, 2018.

[35] N.-M. Ho and W.-F. Wong, "Exploiting half precision arithmetic in Nvidia GPUs," in *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 2017, pp. 1–7.

[36] S. Eliuk, C. Upright, and A. Skjellum, "dMath: A Scalable Linear Algebra and Math Library for Heterogeneous GP-GPU Architectures," *arXiv preprint arXiv:1604.01416*, 2016.

[37] J. L. Gustafson, *The End of Error: Unum Computing*. Chapman and Hall/CRC, 2015.

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

[39] 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.

[40] S. Vakili, J. P. Langlois, and G. Bois, "Enhanced precision analysis for accuracy-aware bit-width optimization using affine arithmetic," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 12, pp. 1853–1865, 2013.

[41] M. Grailoo, B. Alizadeh, and B. Forouzandeh, "Improved Range Analysis in Fixed-Point Polynomial Data-Path," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 11, pp. 1925–1929, 2017.

[42] T. Granlund *et al.*, *GNU MP 6.0 Multiple Precision Arithmetic Library*. Samurai Media Limited, 2015.

[43] C. Kormanyos, *Real-time C++: efficient object-oriented and template microcontroller programming*. Springer, 2015.

[44] D. H. Bailey, H. Yozo, X. S. Li, and B. Thompson, "ARPREC: An arbitrary precision computation package," *Lawrence Berkeley National Laboratory*, 2002.

[45] D. H. Bailey, R. Barrio, and J. M. Borwein, "High-precision computation: Mathematical physics and dynamics," *Applied Mathematics and Computation*, vol. 218, no. 20, pp. 10 106–10 121, 2012.

[46] L. Gerlach, G. Payá-Vayá, and H. Blume, "Efficient Emulation of Floating-Point Arithmetic on Fixed-Point SIMD Processors," in *2016 IEEE International Workshop on Signal Processing Systems (SiPS)*. IEEE, 2016, pp. 254–259.

[47] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning assistant for floating-point precision," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 27.

[48] C. Nguyen, C. Rubio-González, B. Mehne, K. Sen, J. Demmel, W. Kahan, C. Iancu, W. Lavrijsen, D. H. Bailey, and D. Hough, "Floating-point precision tuning using blame analysis," in *International Conference on Software Engineering (ICSE)*. ACM, 2016.

[49] F. De Dinechin, C. Q. Lauter, and G. Melquiond, "Assisted verification of elementary functions using Gappa," in *Proceedings of the 2006 ACM symposium on Applied computing*. ACM, 2006, pp. 1318–1322.

[50] H. Brönnimann, G. Melquiond, and S. Pion, "The design of the Boost interval arithmetic library," *Theoretical Computer Science*, vol. 351, no. 1, pp. 111–118, 2006.

[51] M. Moscato, L. Titolo, A. Dutle, and C. A. Munoz, "Automatic Estimation of Verified Floating-Point Round-Off Errors via Static Analysis," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2017.

[52] A. Abdelfattah, H. Anzt, and A. Bouteiller, "Roadmap for the Development of a Linear Algebra Library," *month*, 2017.

[53] S. Ehsan, A. F. Clark, K. D. McDonald-Maier *et al.*, "Integral images: efficient algorithms for their computation and storage in resource-constrained embedded vision systems," *Sensors*, vol. 15, no. 7, pp. 16 804–16 830, 2015.

[54] K. Ścibisz-Mordelska and R. Nielek, "Lower Precision calculation for option pricing," *Computer Science*, vol. 18, no. 4, 2017.

[55] S. Le Grand, A. W. Götz, and R. C. Walker, "SPFP: Speed without compromise – A mixed precision model for GPU accelerated molecular dynamics simulations," *Computer Physics Communications*, vol. 184, no. 2, pp. 374–380, 2013.

[56] A. Dawson, P. D. Düben, D. A. MacLeod, and T. N. Palmer, "Reliable low precision simulations in land surface models," *Climate Dynamics*, pp. 1–10, 2017.

[57] J. Dongarra, S. Tomov, P. Luszczek, J. Kurzak, M. Gates, I. Yamazaki, H. Anzt, A. Haidar, and A. Abdelfattah, "With Extreme Computing, the Rules Have Changed," *Computing in Science & Engineering*, vol. 19, no. 3, pp. 52–62, 2017.

[58] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, 2015, pp. 1737–1746.

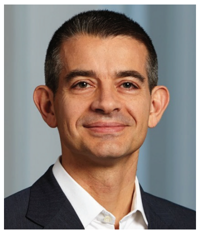
[59] S. Benatti, F. Montagna, D. Rossi, and L. Benini, "Scalable EEG seizure detection on an ultra low power multi-core architecture," in *Biomedical Circuits and Systems Conference (BioCAS), 2016 IEEE*. IEEE, 2016, pp. 86–89.

[60] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with GPUs," *arXiv preprint arXiv:1702.08734*, 2017.

[61] A. Cano, "A survey on graphic processing unit computing for large-scale data mining," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2017.

Giuseppe Tagliavini received the MS degree in Computer Engineering in 2010 and the PhD degree in Electronic Engineering in 2017 from the University of Bologna. He is currently a post-doctoral researcher at the Department of Electrical, Electronic and Information Engineering (DEI) at the University of Bologna. His research interests include programming models and run-time optimization for many-core accelerators, software design for low-power embedded systems and advanced frameworks (compilers, runtime environments, exploration tools) for emerging computing architectures.

Andrea Marongiu received the MSc degree in electronic engineering from the University of Cagliari, Italy, in 2006 and the PhD degree in electronic engineering from the University of Bologna, Italy, in 2010. Since 2013 he has been a Research Fellow at ETH Zurich. He currently is an Assistant Professor at the University of Bologna. His research interests concern parallel programming model and architecture design in the single-chip multiprocessors domain, with special emphasis on compilation for heterogeneous architectures, efficient usage of on-chip memory hierarchies and SoC virtualization. He has published more than 100 papers in peer reviewed international journals and conferences. He is a member of the IEEE.



Luca Benini holds the chair of Digital Circuits and Systems at ETHZ and is Full Professor at the University of Bologna. Dr. Benini's research interests are in energy-efficient system design for embedded and high-performance computing. He is also active in the area of energy-efficient smart sensors and ultra-low power VLSI design. He has published more than 800 papers, five books and several book chapters. He is a Fellow of the IEEE and a member of the Academia Europaea. He is the recipient of the 2016 IEEE CAS Mac Van Valkenburg award.