Towards a Transprecision Polymorphic Floating-Point Unit for Mixed-precision Computing

*

Alisson Linhares
Institute of Computing
University of Campinas (Unicamp)
alisson.carvalho@students.ic.unicamp.br

Rodolfo Azevedo
Institute of Computing
University of Campinas (Unicamp)
rodolfo@ic.unicamp.br

***Abstract*—Mixed-precision is a paradigm that tries to combine computations with different levels of precision to compose results. This approach has been used extensively to optimize scientific applications and has shown speed and energy gains, without causing any relevant precision loss. However, to exploit mixed precision opportunities most applications need to be recompiled to use different instructions and types. Thus, in this work, we present a new floating-point unit design, able to automatically decide when an instruction should be executed using less precision, without recompilation or user direct intervention. Our proposal takes advantage of ad-hoc polymorphism to perform computations with different data types, dynamically selecting a proper instruction on demand, and can also be configured according to overall precision requirements. Our simulated results show that, for some double-precision benchmarks, we are able to execute more than 90% of all floating-point operations in half-precision, without affecting its accuracy and resulting in a precision error below 1%. In addition, this new technology may increase instruction level parallelism and cut down the necessity for type casting operations.**

***Index Terms*—Transprecision, Mixed-Precision, Floating-Point Unit, RISC-V**

I. Introduction

In traditional computing systems, most operations are carried out in a conservative way - providing the largest available precision and accuracy whenever possible. Hence, from the hardware perspective, it does not matter whether one or sixty-four bits are enough to offer a valid result. If an instruction was designed to use 64-bit precision, it will be executed as if 64-bit precision was strictly required, without worrying about data distribution nor the application minimal requirements.

Contradicting this traditional paradigm, several studies have shown that modern hardware is underutilized. Integer operations, for example, rarely use their full dynamic range, requiring, most of the time, only a small fraction of all allocated resources to generate same results [1]–[3]. Moreover, skipping computations or performing inexact calculations can result in considerable speed and energy efficiency gains, justifying a little accuracy loss [4].

In an attempt to improve floating-point hardware utilization, researchers have combined different levels of precision in order to produce accurate results. This paradigm, also known as mixed-precision computing, has shown speed and energy gains without relevant precision loss [5]–[9]. However, detecting which operation can be performed using less precision, avoiding negative effects on the accuracy and performance is not an easy task.

For most applications, the amount and complexity of floating-point operations makes mixed-precision optimization impractical. In addition, computations with different types require casting instructions to convert values to a common format. Thus, in some situations, even though most variables can be represented using a less precise type, the increase in the amount of type casts inside loops can result in a slowdown.

Different from other paradigms, the transprecision-computing is an extension of approximated-computing and a more general solution than mixed-precision. It attempts to combine the advantages of approximated techniques with the accuracy of traditional systems, providing ways to guarantee that the minimal precision constraints on the final results are always met [10].

Once intermediate computations can usually be performed with less precision and without accuracy loss, a transprecise hardware can be designed to use approximated circuits whenever possible, alternating the computation to a more precise implementation on demand. In addition, a transprecision solution integrates hardware and software, offering granular control of the approximated hardware and feedback information to upper layers. This design allows better system utilization, once application can keep track of the overall error and dynamically decide if more precision is required.

In this work, we propose a transprecision design to improve floating-point hardware utilization. Our proposal takes advantage of hardware ad-hoc polymorphism to perform computations with different data types, dynamically selecting an appropriate precision on demand. Our hypothesis is that these hardware changes will increase instruction level parallelism

This work was executed with support from CAPES (Coordination for the Improvement of Higher Education Personnel), grant #2013/08293-7 from FAPESP (São Paulo Research Foundation) and grants #309794/2017-0 and #438445/2018-0 from CNPq (National Council for Scientific and Technological Development).

978-1-7281-4194-7/19/$31.00 ©2019 IEEE
DOI 10.1109/SBAC-PAD.2019.00022

and cut down the necessity of type casting instructions for floating-point operations.

This paper is organized as follows: in Section II, we present a literature review; in Section III, we present the theoretical design of our TFPU (Transprecision Floating-point Unit); in Section IV we present our infrastructure for floating-point hardware exploration; in Section V, we present our results, including the potential speed-up of this new technology; in Section VI we explain how this new technology can improve floating-point hardware and what we can do in the future; Lastly, in Section VII, we present our conclusions.

II. Related work

Researchers have demonstrated gains in speed and memory efficiency by changing double-precision variables into less precise types. In this section, we summarize the most relevant contributions to the field [5].

A. Software-assisted mixed-precision tuning

The first attempt to automatically detect mixed-precision opportunities in floating-point computations was proposed by Michael O. Lam et al. (2013) [6], [11]. They designed a framework, called CRAFT HPC, that uses binary instrumentation and modification to build mixed-precision configurations of binaries that were originally designed to use only double-precision.

The CRAFT HPC uses an automatic breadth-first search algorithm to find code regions that can successfully be replaced with single-precision. Unfortunately, this technique requires an analysis of 2^n different configurations, where n is the number of floating-point instructions. This worst-case scenario makes a brute-force solution impractical for a large code base, since an evaluation of a test configuration requires a full program run.

In an effort to overcome the search space limitation and improve floating-point mapping, different solutions were proposed [8], [12]–[14]. However, the downside of most floating-point mixed-precision tools is the fact they focus on improving performance and energy efficiency for specific inputs using static error control. Unfortunately, these approaches offer no guarantee that error requirements will be met across all program inputs. To overcome this limitation, different rigorous alternatives have been explored [9], [15], [16], but the speed up improvement are arguably small, when compared against more advanced hardware solutions and previous publications.

B. Hardware-assisted mixed-precision tuning

Many experiments show potential on the usage of less accurate floating-point types [17]. Following this trend, companies have introduced half-precision floating-point support into their hardware. Modern NVIDIA GPUs, for instance, can join half-precision operations and execute them in parallel using single-precision units. This action can also be specified manually, using the new half2 vector data type.

According to recent publications, training deep neural networks with NVIDIA's half-precision support can offer up to 8x more arithmetic throughput when compared to single-precision [18]. However, properly conversion of single-precision applications to half precision is a problematic task. To simplify this process, Nhut-Minh Ho et al. (2017) [19] has developed a tool built on LLVM Clangs LibTooling, called cuda-half2.

Cuda-half2 is able to rewrite CUDA code and exploit the native half-precision operations. However, by default, this tool converts all floating-point variables to half and half2. This straightforward conversion can lead to execution errors and a drastic reduction in precision and accuracy - once half precision has a dynamic range of 2^{-15} to 2^{14} with 10-bit of mantissa, while double has a range of 2^{-1024} to 2^{1023} with 52-bit of mantissa. Thus, different from most mixed-precision tuners, this solution works as a guide, letting the developer decide which set of variables to optimize.

Similar to NVIDIA, Intel researchers have presented a floating-point unit that performs FMA (Fused Multiply-Adds) operations with automatic precision tracking [20], [21]. To increase throughput and improve energy efficiency, the proposed FMA exploits vectorization and can configure its single-precision hardware to operate in three different modes: 1-way (using 24 bit mantissa), 2-way (using 12 bit mantissa) or 4-way (using 6 bit mantissa). A certainty tracking circuits operate in parallel with exponent computation, calculating operand-dependent accuracy bounds that indicate the need for increased precision. However, there is no public information if similar techniques were incorporated in current Intel hardware.

Aligned with our proposal, Albert Ou et al. (2014) [22] added a hardware support for mixed-precision operations in a decoupled vector-fetch data-parallel accelerator, called Hwacha. Hwacha was enhanced with a configuration instruction (vsetfg) and operations on packed data. The configuration instruction was designed to set the number of architectural registers and their individual data type widths, by merging subsequent positions according to configuration requirements. The Load operations can fetch blocks of data and populate those configured registers according to the data type. Because the number of elements inside a register can be configured, wide registers can be cast to smaller ones, improving the performance of mixed-precision operations.

According to their experiments, the support for mixed-precision processing in Hwacha results in 5.8% area overhead and a speed up as high as 62.1% for a double-precision baseline [22]. However, this solution was strictly designed for a vector processor and requires the application to be rewritten to exploit this hardware support.

Recently, Tagliavini et al. (2018) [7] accomplished significant energy savings by incorporating non-standard sub-32-bits FP formats. Using a library called FlexFloat to encode non-standard type and the fpPrecisionTuning tool to search for the best mixed-precision configuration. Their experimental results show that up to 90% of FP operations could be safely scaled down to 8-bit or 16-bit formats. Moreover, by leveraging vectorization, the execution time was decreased by 12% and memory accesses were reduced by 27% on average, leading

to a reduction of energy consumption up to 30%.

Tagliavini et al. (2018) also presented a transprecision FP unit extension for PULPino SoC [23]. This TFPU has support for two non-standard floating-point formats (smallfloats), that were explored in more details in their previous publications [7]. Their results show that, combining the hardware support for SmallFloat with fpPrecisionTuning improved the system performance by 15% to 25% and the energy efficiency by 14% to 18%.

C. Polymorphic operations in hardware

In modern programming languages, most arithmetic operations mean different things according to the types of their operands. Take as an example the expression below:

$$100 + 200 = 300 \tag{1}$$

$$1.2 + 1.3 = 2.5 \tag{2}$$

In this example, the plus sign will have a different behavior depending on the type specified, requiring different instructions for each expression. Ambiguous operators of this sort are polymorphic (Ad hoc polymorphism) as they can have several forms depending on their arguments.

Channoh Kim et al. (2017), proposed the implementation of polymorphic instructions in hardware to accelerate script languages [24]. This new architecture calculates and checks the dynamic type of each variable implicitly in hardware, rather than explicitly in software. Thus, based on the TAG information in the register file, the CPU is able to automatically decide which unit should perform the correct operation. Furthermore, according to experimental results, this new architecture achieves speed up of 11.2% for a JavaScript scripting engine and 9.9% for Lua engine, demonstrating an interesting potential in this technology.

III. TFPU

As explained in Section I, our main goal is to extend a RISC-V floating-point unit with polymorphic operations and transprecision capabilities. By default, the RISC-V ISA provides a set of 32 floating-point registers that can be accessed in 32-bit (F extension), 64-bit (G extension) and 128-bit (Q extension) mode. Encoded in the floating-point instruction, there is a fixed bit field called FMT, that control if the computation uses single-precision, double-precision or quad-precision data. This design wastes part of the encoding space that could be useful for others extensions - once the single-precision, double-precision and quad-precision instruction set are practically identical, differing only in the FMT field.

Our proposal unifies all floating-point operations into one group of polymorphic instructions, optimizing the encoding space. These instructions use the register content to decide the precision, eliminating the need for cast operations, such as `fcvt.s`, `fcvt.d`, and `fcvt.q`.

An overview of the TFPU architecture is presented in Figure 1. In this example, we show the register file state after executing 5 instructions in a scalar pipeline. The *addx* and *mulx* are polymorphic operations that use the type stored in each register to generate results; the value 1.5f, 3.25f and 69332f are stored in memory. Once 1.5 and 3.25 can be represented in half precision without loss, after a load operation, these values are tagged with HP, allowing future computations to be performed using less precise units.

The TFPU is divided into three main modules: PRF (Polymorphic Register File), PCTRL (Polymorphic Control Logic) and TPU (Type Promotion Unit). The goal of PCTRL and PRF is to enable polymorphic computations without changing the LSU (Load Store Unit) and FPU (Floating-point Unit) pipelines. While the TPU add support for transprecision and approximated computing. In the following subsections, we explain each part of our design with more details.

A. Polymorphic Register File

Different from a classic register file implementation, a polymorphic register file stores the type and format of each register in a tag field. In our design, the tag is used by the PCTRL to select an appropriated precision to perform computations.

The load instruction configures the tag automatically. For example, when the CPU loads a single-precision value from memory, using FLS instruction, it marks the register content with an SP tag. Analogously, when a double-precision is loaded using FLD, the tag value is set to DP. It is also possible to configure the tag manually. We added a *cast* instruction that converts the register's content to a different format, changing the precision and the data representation.

B. Polymorphic Control Logic

The PCTRL (Polymorphic Control Logic) is responsible to convert the register content to a common format. It uses a best-fit policy algorithm (BFP) to decide an appropriated execution unit for each operation.

The BFP algorithm makes decisions based on the tag stored in the register file. If two values are marked with the same tag, no casting is required. However, if one value is tagged with a more precise type or different representation, the PCTRL casts the less precise type to a common format, then sends both values to an appropriated execution pipeline.

The conversion between types is lightweight and straightforward. We expand the exponent field, without performing any round-off operation, and set unnecessary bits to zero.

C. Type promotion unit

A floating-point number has two fundamental parts: one that is directly related to accuracy and other that is directly related to the precision. To better explain how transprecision can affect the accuracy, take as an example the number 2^{126}. This number can be represented without any loss in a single-precision format. However, once half-precision has only 5 exponent bits, the largest non-infinity value that can be represented using half-precision is 2^{16}. Thus, any half-precision computation that can extrapolate the range between 2^{15} and 2^{-14} could potentially decrease the accuracy. In addition, due to a reduction in representation range, some operations may

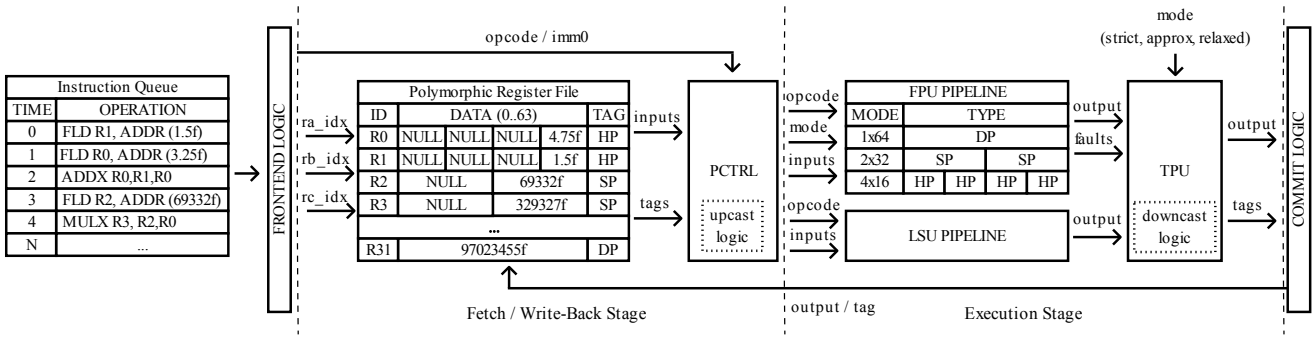


Fig. 1: Proposed TFPU design.

result in an underflow, overflow, positive infinity, negative infinity, not a number or division by zero.

The TPU (Type promotion unit) is responsible for normalizing results, selecting an appropriate tag for each output and intercepting IEEE 754 fault signals. Additionally, it protects the system against the propagation of highly inaccurate results, such as NaN, -INF and +INF.

The TPU takes different actions according to each IEEE 754 signal. When the execution unity produces an underflow signal, the TPU upgrades the precision, clears the mantissa and sets the exponent to the smallest possible value that is not compatible with the previous tag. For example, if after a half-precision operation an underflow happens, the TPU promotes the type to single and changes the exponent to -15, once the smallest representable exponent value in half-precision is -14.

When an infinity or overflow happens, the TPU upgrades the precision and sets the mantissa to the maximum value. It is also necessary to fix the exponent, setting it to the maximum value that is not compatible with the previous tag. For example, if after a single-precision operation an overflow or infinity happens, the TPU promotes the type to double (DP) and changes the exponent to 128, once the biggest exponent value that can be represented in single-precision is 127.

The TPU uses the inexact signal to decide if we need to promote the tag or stay in the same precision. If a zero signal is received and the inexact signal is low, we downgrade the tag. On the other hand, if an inexact signal is high and zero signal is low, we upgrade the tag.

To illustrate how the inexact signal affects the precision, suppose the TFPU needs to add two values, 32768 and 0.5. Both values can be represented using half-precision without any representation loss. However, if we add these numbers using an IEEE 754 half-precision compatible unit, the result will be 32768, instead of 32768.5 and the inexact signal will be high. In this situation, future computations involving this result have a high potential of increasing the accumulated error especially if the result is used in expressions inside a loop.

When we use the inexact signal to guide promotions, an addition of 32768 and 0.5 will continue to be 32768 instead of 32768.5, but tagged as single-precision instead of half-precision. The tag forces subsequent computations involving the result to use more precise execution units.

D. Extending the RISC-V instruction set

We extended the RISC-V ISA with 6 new instructions (flh, fsh, shpt, sspt, sdpt and cast). In addition, we also changed the behavior of loads, stores and all arithmetic operations - adding support for TAG information. The Table I has a summary of all changes.

Instruction	Syntax	Description
flh	**flh** fd, [*addr*]	Loads 2 bytes and sets the TAG to HP.
flw	**flw** fd, [*addr*]	Loads 4 bytes and selects the best compatible TAG.
fld	**fld** fd, [*addr*]	Loads 8 bytes and selects the best compatible TAG.
flq	**flq** fd, [*addr*]	Loads 16 bytes and selects the best compatible TAG.
shpt	**shpt** *imm8*	Configures a transition threshold for HP.
sspt	**sspt** *imm8*	Configures a transition threshold for SP.
sdpt	**sdpt** *imm8*	Configures a transition threshold for DP.
cast	**cast** fd, *tag*	Casts the register content to another format and sets the tag.
fsh	**fsh** fd, [*addr*]	Stores the half-precision value into memory.
fsw	**fsw** fd, [*addr*]	Converts value to single and stores it into memory.
fsd	**fsd** fd, [*addr*]	Converts value to double and stores it into memory.
fsq	**fsq** fd, [*addr*]	Converts value to quad and stores it into memory.

TABLE I: Transprecision instruction set.

For compatibility reasons we maintained the same RISC-V opcodes. However, there is no difference in using a double, single or quad instruction. When the transprecision hardware is enabled, the TFPU decides what to do based on the tag information, instead of using the FMT field.

The *shpt*, *sspt*, *sdpt* and *spqt* are instructions designed to manually configure the transition between data types. This threshold can be used by future TFPU implementations to decide when it is an appropriated moment to change internal data representation or increase precision.

By default, all operations in the range between 2^{15} and 2^{-14} is tagged as half-precision and any value outside this range is tagged with a higher precision. The *shpt*, for example, can be used to configure different ranges such as 2^{8} and 2^{-8} forcing numbers outside this scale to be mapped to a more precise type. In Section V we show what happens when we use different thresholds to control transitions.

IV. Simulation Infrastructure

In order to estimate the potential of our TFPU design, we have implemented an object-oriented support for RISC-V floating-point units inside the Spike emulator. Our implementation abstracts the Spike's internal details, providing a default class (DefaultBehavior) that can be inherited and modified as needed. Figure 2 presents an overview of our simulation environment.

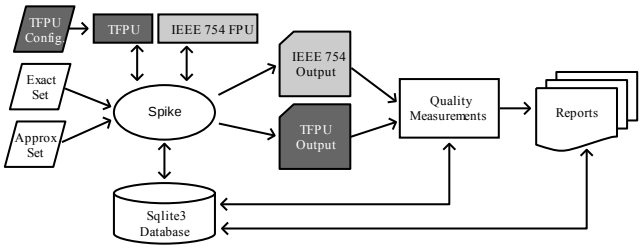


Fig. 2: Simulation environment

Integrated with Spike, we also built an analysis infrastructure that automatically runs a set of 37 benchmarks, collecting execution statistics and saving them in a database for further verification.

We divided all benchmarks into two sets: approximated and exact. The approximated set was used to measure the mean relative error of each execution. These benchmarks generate results that can be analyzed in order to determine the overall precision and accuracy of each FPU implementation; the exact set has more sophisticated benchmarks, that either perform a self-test on the result or generate a right or wrong answer.

At the end of Spike execution, our infrastructure compares the TFPU results with the statistics of the original IEEE 754 FPU. Based on the information stored in the database and quality measurements, we can use SQL language to extract a variety of useful statistics, including: exponent distribution; instruction frequency; read and write access; total of type promotions and demotions; total of zeros, infinities, overflows, underflows, subnormal and normal computations; and many others. Moreover, our infrastructure also generates a variety of charts, in a paper ready format, simplifying the data visualization.

V. Results

We evaluated our proposal using applications from several different suits, including PARSEC, CoreMark Pro, NAS, MiBench, axbench, linuxbench and The Computer Language Benchmarks Game. From each suit, we choose only applications that contain double-precision operations. The complete list of benchmarks can be viewed in Table II.

Most approximated benchmarks were taken from related works. This is the case of GSL and axbench sets: the GSL set contains programs from Precimonious project, while axbench is used for research involving approximate computing. Beyond that, some applications were adapted to output computation results, once by default, some applications print only the running time or unnecessary information. This was the case of fbench, linpack, n-body, fluidanimated and basicmath.

It is important to point out that some benchmarks used by Precimonious are dependent of the obsolete Intel FPU 8087 supports, since the input files were encoded using 80-bit long doubles types, rather than 128-bit quad-precision. Thus, we modify all input files and all 8087 dependent codes - including the ones inside the cov_serializer - to an IEEE 754 compliant version. In addition, the results of all modified gsl benchmarks

	Benchmark	Description
Approximated set	arclength	Estimates the arc length of the function over the interval (0, pi).
	simpsons	Uses Simpson's Rule for approximating the integral of a function between two limits.
	gsl-sum	Computes the extrapolated limit of series using a Levin u-transform.
	gsl-roots	Finds roots of arbitrary one-dimensional functions.
	gsl-polyroots	Evaluates a polynomial with real coefficients for complex variables.
	gsl-gaussian	Cumulative distribution functions.
	gsl-fft	Computes complex Radix-2 forward fast Fourier using gsl.
	gsl-blas	Operations on vectors and matrices.
	gsl-bessel	Bessel function computation.
	fbench	Optical design raytracing algorithm.
	linpack	Solves a dense system of linear equations.
	basicmath	Performs a variety of mathematical calculations.
	mandelbrot	Rendering a Mandelbrot fractal using a static number of iterations.
	spectralnorm	Calculate an eigenvalue using the power method.
	nbody	Model the orbits of Jovian planets.
	swaptions	Pricing of a portfolio of swaptions.
	blackscholes	Option pricing with Black-Scholes Partial Differential Equation.
	fluidanimate	Fluid dynamics for animation purposes with Smoothed Particle Hydrodynamics method.
	fft	Radix-2 Colley-Tykey fast Fourier implementation.
	inversek2j	Inverse kinematics for 2-joint arm.
	jpeg	JPEG encoding.
	jmeint	Triangle intersection detection.
	c-ray-f	Renders a complex 3D scene.
Exact set	canneal	Simulated cache-aware annealing to optimize routing cost of a chip design.
	fasta	Generates DNA sequences.
	linear-alg	A linear algebra routine derived from LINPACK.
	loops	Variety of kernels based on Livermore loops.
	nnet	Use a neural net to evaluate patterns.
	radix2	This benchmark performs FFT with Radix2 on the input.
	ffbench	Fast Fourier transform of a square matrix of complex numbers.
	is.W.x	Integer Sort (uses a complex double-precision number generator).
	ep.W.x	Embarrassingly Parallel.
	mg.W.x	Multi-Grid on a sequence of meshes.
	ft.W.x	Discrete 3D fast Fourier Transform.
	bt.W.x	Block Tri-diagonal solver.
	sp.W.x	Scalar Penta-diagonal solver.
	lu.W.x	Lower-Upper Gauss-Seidel solver.

TABLE II: List of benchmarks used

are equivalent to the original results, once there was no need to use quad-precision, since all applications were casting types to single during initialization of their kernels.

The precision error analysis methodology is based on axbench. We used the Mean Absolute Percentage Error (MAPE) to determine the quality of the numerical results and the Root Mean Square Error (RMSE) for images. The errors were normalized on a scale of 0 to 1. Where 0 represents the minimum error in relation to the original program and 1 is the maximum possible error. In case of execution failures, even if most of the result is correct, errors are rounded to 1.

A. Tested Configurations

There are different ways to configure our TFPU. For this exploration, we tested our design using 32 different setups, described below:

- HP TFPU: This TFPU executes all operations using half-precision. Whenever a value is loaded from memory, it is automatically converted to 16 bits. Similarly, when we need to store data into memory, the data is converted back to the original precision, using the FMT field as a guide. This FPU is equivalent to change all double-precision and single-precision operations for half-precision. For a TFPU with half-precision support, executing everything in half-precision with 0 error would be the best result in terms of performance.
- SP TFPU: This TFPU is similar to the HP TFPU, but it uses single-precision instead of half-precision. Thus, if an application requires only floats (single-precision), the result generated by this FPU will be identical to an IEEE 754 implementation.
- Strict TFPU (ST): This TFPU is implemented according to the proposed architecture in Section III. Whenever a

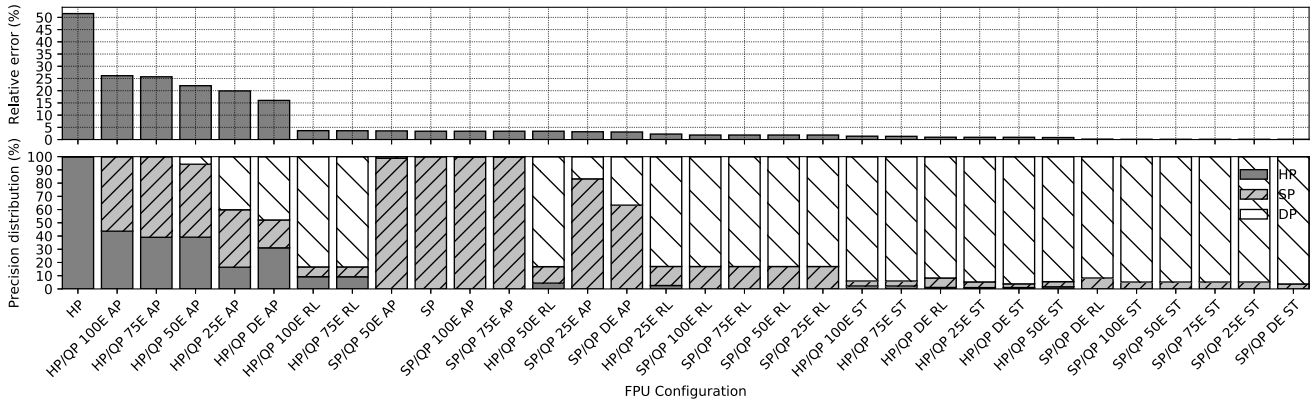


Fig. 3: Precision distribution and relative error of each FPU configuration.

load or casting occurs, the data is converted to a smaller format, capable of representing the value without loss. This TFPU is extremely conservative, promoting the type whenever an inexact signal is generated by the execution unit.

- Approximated TFPU (AP): This TFPU uses only the dynamic range to promote or demote types. Therefore, it tends to maximize the number of computations using half-precision, changing the format only when it is not possible to represent the value within a smaller range.
- Relaxed TFPU (RL): This TFPU is similar to the ST TFPU. The only difference is that it does not use the inexact signal to promote types - promotions are solely guided by the mantissa and transition threshold.

We also tested each configuration using five thresholds and two precision ranges. The DL threshold is standard range. The E25, E50, E75 and E100 are the proportions used to define the transition of each type. For example, with E25 the TFPU will tag values between 2^4 and 2^{-4} (25% of 16) to half-precision; 2^{32} and 2^{-32} (25% of 128) to single-precision; and 2^{256} and 2^{-256} (25% of 1024) to double-precision.

We tested only two precision ranges: SP-QP and HP-QP. SP-QP allows transitions between single-precision and quad-precision. Similarly, HP-QP allows any transition between half-precision and quad-precision.

B. Approximated results

In Figure 3 we show the precision distribution and the mean error of all approximated benchmarks. All errors were normalized between 0 and 1 to avoid interference, considering that some benchmarks had errors above 100%. As expected, the approximated configuration can execute more operations in half and single precision than the strict and relaxed versions.

Without compiler support and user guidance, the strict and relaxed configurations were unable to produce relevant results, once most operations are still executed in double-precision. However, it is important to notice that most strict and relaxed configurations exhibit negligible errors in most situations - with errors below 0.01%.

According to our experiments, the variation in the exponent transition can have a relevant impact in error and performance. Thus, we believe it is interesting to let the user control the precision by changing these parameters according to the application dynamic range.

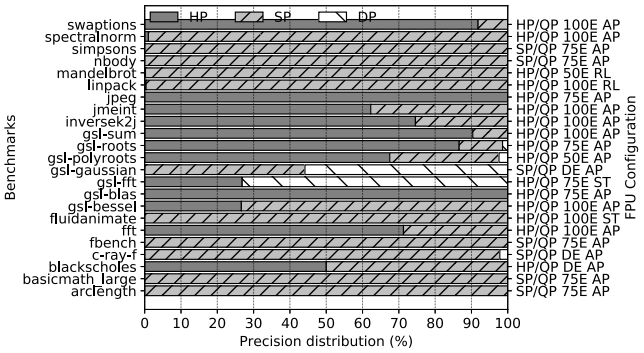


Fig. 4: Best results with less than 5% error.

In Figure 4 we show the results of the best configuration for each approximated benchmark. To create this graphic we considered all FPUs, including the SP, HP and the original IEEE 754 FPU. The ranking was calculated based on the theoretical throughput of a shared 64 bit vector unit - considering that we can execute in three different modes, like modern CPUs: 4x16, 2x32 and 1x64.

To discard highly inaccurate results, we set the maximum acceptable error to 5%. As we can see, depending on the TFPU configuration, is possible to improve the theoretical throughput significantly. All benchmarks had mapping improvements, without causing any relevant accuracy loss or increasing instruction count.

The best results were found in jpeg and gsl-blass, They were able to execute almost 100% of all double-precision instructions in half-precision, with errors below 0.1%. The worst result was gsl-gaussian, that mapped about 44% of all double-precision instructions to single-precision.

C. Exact results

To measure how close our execution is to the original IEEE 754 FPU, we ran a set of benchmarks that performs self verification. Some of them are highly sensible to precision variation.

The is.W.x benchmark, for example, uses a double-precision variable to generate pseudo aleatory numbers. Therefore, any alteration in the mantissa, can lead to a different result.

In addition, some benchmarks perform checksums on results or measure the RMSE distance of the output. As we are losing some precision by executing operations in less precise types, the transprecision will also affect the RMSE calculation, leading to a threshold very close to the original, but not equal.

Naturally, we do not expect our solution to work with every application, since we are not following the IEEE 754 guide. However, we expect the application behavior to be very close to an execution using IEEE 754 FPU.

	HP	SP	HP/QP AP 25E	HP/QP RL 25E	HP/QP ST 25E	HP/QP AP 50E	HP/QP RL 50E	HP/QP ST 50E	HP/QP AP 75E	HP/QP RL 75E	HP/QP ST 75E	HP/QP AP 100E	HP/QP RL 100E	HP/QP ST 100E	HP/QP DE AP	HP/QP DE RL	HP/QP DE ST	SP/QP AP 25E	SP/QP RL 25E	SP/QP ST 25E	SP/QP AP 50E	SP/QP RL 50E	SP/QP ST 50E	SP/QP AP 75E	SP/QP RL 75E	SP/QP ST 75E	SP/QP AP 100E	SP/QP RL 100E	SP/QP ST 100E	SP/QP DE AP	SP/QP DE RL	SP/QP DE ST
linearalg		✓			✓	✓		✓	✓		✓	✓		✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ft.W.x																																
mg.W.x					✓	✓										✓	✓		✓	✓		✓	✓		✓	✓			✓	✓	✓	✓
sp.W.x																																
is.W.x																																
ep.W.x					✓	✓		✓	✓		✓	✓		✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
lu.W.x																			✓	✓		✓	✓		✓	✓		✓	✓		✓	✓
bt.W.x																																
canneal		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ffbench			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
fasta		✓		✓	✓		✓	✓		✓	✓		✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
loops-all				✓	✓		✓	✓								✓	✓	✓	✓	✓		✓	✓		✓	✓		✓	✓	✓	✓	✓
nnet																																
radix2																																
Successes	0	3	2	7	7	2	6	6	2	5	5	2	5	5	2	7	7	5	8	8	4	8	8	4	8	8	4	8	8	5	8	8

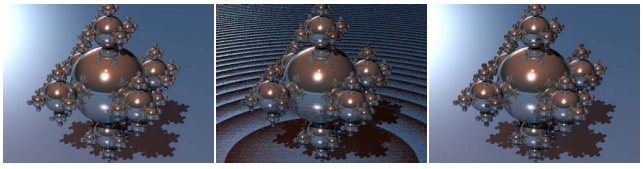
TABLE III: Exact results

In Table III we present our results in a set of benchmarks that performs self checking. It is important to point out a failure does not mean the application generated a wrong result. It is just that the result is neither inside a very restrictive precision requirement or equal bitwise.

Our approximated configuration achieved positive results in most situations. In ffbench, for example, we were able to map 98.45% of all double-precision variables to half-precision, passing in all self validation tests. Yet, even in this benchmark, that uses a more relaxed error verification, if we try to use the SP or HP configurations without performing transitions between types, the application is going to report hundreds of faults in the result.

Based on this experiment, we can conclude that our transprecise solution is safer than converting all double-precision variables to single or half, once our TFPU was able to succeed in situations where the SP and HP FPU did not.

D. Case study



(a) double-precision (b) single-precision (c) trans-precision

Fig. 5: Results of three different executions of c-ray-f bench.

In Figure 5 we have a 3D fractal rendered using three different executions of c-ray-f benchmark. The Figure 5a is the result of the original c-ray-f using an IEEE 754 FPU. The Figure 5b we present the result of the single-precision version of c-ray-f also running in an IEEE 754 FPU. The Figure 5c shows the result of the double precision version of c-ray-f using the SP/QP AP DE configuration.

Based on our experimental results, c-ray-f uses a dynamic range between 2^{49} to 2^{-1023}, but 99.99% of all computations happens in a number scale between 2^{127} and 2^{-126}. This means that, during the majority of the execution time, the FPU uses less than 8 bits of exponent, wasting at least 3 bits of the 11 bits exponent encode. Even thought a substantial part of the computation could be performed using float data types, if we try to run this application using single-precision variables it will result in 29.3% RMSE in relation to the original double-precision version, generating the image presented in Figure 5b.

Once our TFPU only performs casting operations when the data can fit in the target register, the SP/QP AP DE configuration was able to execute 97.8% of all instructions in single-precision, resulting in an image with 3.6% error. In the Figure 5c, we see an image almost identical to the original, but it uses less hardware resources than the double-precision version and is 8 times more precise than the single-precision version. Moreover, our TFPU was able to automatically optimize the original double-precision application, without any human intervention.

VI. Discussion

According to our investigation, CRAFT HPC [11] is a mixed-precision tuning framework capable of reaching an optimal floating-point balance, without hardware support. However, as discussed in Section II-A, its searching technique is not practical in real life applications, due to a high running time. PROMISE [12], Precimonius [13] and fpPrecisionTuning [14], on the other hand, are faster than CRAFT HPC, but they provide small gains. Daisy [16] and FPTune [9] are the only tools capable of guaranteeing a minimal error control, but the speed up improvements are arguably small, when compared against hardware solutions. The Hwanch [22] and cuda-half2 [19] can achieve higher speed ups, but they are not designed for general purpose CPUs. Finally, we have the Tagliavini's solution [23], a hardware-assisted infrastructure for a general purpose CPU, that requires compilation and profiling to properly work.

Our proposal differs from previous publications in four fundamental ways: I. it explores the fact that precision requirements vary during each computation; II. it makes possible to better use hardware resources by changing precision requirements during runtime; III. It eliminates the need for casting operations between floating-point types; and most importantly, IV. This is the first attempt to design a Tranprecision FPU with polymorphic capabilities.

It is important to notice that our solution does not require direct changes in the source code to work. However, the compiler can be modified to generate binaries to exploit our transprecision and polymorphic characteristics. As an example, a mixed-precision application could remove all casting

operations, letting our TFPU decide the correct precision - solving an important limitation of most mixed-precision software proposals. In addition, we can combine our solution with tools like fpPrecisionTuning [14], Precimonius [13] or FPTuner [9] to improve memory access and space.

Currently, we are implementing our TFPU design in The Berkeley Out-of-Order Machine (BOOM) to obtain more advanced statistics, such as area, maximum operation frequency and energy consumption. We believe that our modifications have the potential to improve performance, once many publications have demonstrated energy and speed up improvements on joining half-precision operations together and executing them in parallel using single-precision units [7], [19]–[21].

In the future, we intend to use our simulation environment to test different hardware changes to improve performance, including: adding a precision predictor hardware, to decide the best moment to promote types; and an intermediate floating-point representation, to accelerate type conversions.

VII. Conclusion

In this paper, we presented the first steps toward a polymorphic floating-point unit with transprecision capabilities. Unlike previous works, our optimization is done at runtime, without requiring programs to be recompiled or profiled in order to decide an appropriated precision. In addition, our TFPU is very simple and hardware friendly - remapping only controls and data signals to modify the floating-point pipeline behavior.

In order to evaluate our proposal, we have developed a sophisticated simulation environment on top of Spike. The support infrastructure runs automatically a set of 37 double-precision benchmarks, collecting detailed statistics of the FPU. This new infrastructure will help future floating-point hardware and software explorations, simplifying approximated-computing and transprecise investigations.

According to our experimental results, our design is able to perform a substantial part of all double-precision operations in single-precision and half-precision, without compromising the application execution flow. In some benchmarks we were able to execute more than 95% of all double-precision operations in half-precision, resulting in less than 5% error. In addition, we demonstrated that by using exception signals, our solution can even pass in benchmarks that perform self-checking, achieving mapping results close to related works.

References

[1] D. Brooks and M. Martonosi, "Dynamically exploiting narrow width operands to improve processor power and performance," in *Proceedings Fifth International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 13–22, Jan 1999.

[2] M. Imani, S. Patil, and T. Rosing, "Dcc: Double capacity cache architecture for narrow-width values," in *2016 International Great Lakes Symposium on VLSI (GLSVLSI)*, pp. 113–116, May 2016.

[3] M. K. Tavana, S. A. Khameneh, and M. Goudarzi, "Dynamically adaptive register file architecture for energy reduction in embedded processors," *Microprocessors and Microsystems (MICPRO)*, vol. 39, pp. 49–63, Mar. 2015.

[4] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate computing: A survey," *IEEE Design Test*, vol. 33, pp. 8–22, Feb 2016.

[5] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov, "Accelerating scientific computations with mixed precision algorithms," *Computer Physics Communications (CPC)*, pp. 2526–2533, 2009.

[6] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. Legendre, "Automatically adapting programs for mixed-precision floating-point computation," in *Proceedings of the 27th International ACM Conference on Supercomputing (ICS '13)*, p. 369, ACM Press, jun 2013.

[7] G. Tagliavini, S. Mach, D. Rossi, A. Marongiu, and L. Benin, "A transprecision floating-point platform for ultra-low power computing," in *DATE*, pp. 1051–1056, March 2018.

[8] C. Gonzlez, C. Nguyen, H. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning assistant for floating-point precision," in *Supercomputing Conference (SC)*, pp. 1–12, Nov 2013.

[9] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić, "Rigorous floating-point mixed-precision tuning," *SIGPLAN Not.*, vol. 52, pp. 300–315, Jan. 2017.

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

[11] M. O. Lam and J. K. Hollingsworth, "Fine-grained floating-point precision analysis," *Int. J. High Perform. Comput. Appl.*, vol. 32, pp. 231–245, Mar. 2018.

[12] S. Graillat, F. Jézéquel, R. Picot, F. Févotte, and B. Lathuilière, "Auto-tuning for floating-point precision with Discrete Stochastic Arithmetic." working paper, June 2016.

[13] C. R. Gonzlez, C. Nguyen, 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 *38th International Conference on Software Engineering (ICSE)*, pp. 1074–1085, May 2016.

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

[15] J. Lee, H. Vandierendonck, M. Arif, G. D. Peterson, and D. S. Nikolopoulos, "Energy-efficient iterative refinement using dynamic precision," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, pp. 1–1, 2018.

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

[17] P. Micikevicius, S. Narang, J. Alben, G. F. Diamos, E. Elsen, D. García, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," *CoRR*, vol. abs/1710.03740, 2017.

[18] L. Durant, O. Giroux, M. Harris, and N. Stam, "Inside volta: The worlds most advanced data center gpu," *https://devblogs.nvidia.com/inside-volta/*, May 2017.

[19] N. Ho and W. Wong, "Exploiting half precision arithmetic in nvidia gpus," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, Sept 2017.

[20] 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 *2012 IEEE International Solid-State Circuits Conference*, pp. 182–184, Feb 2012.

[21] H. Kaul, M. A. Anders, S. K. Mathew, R. K. Krishnamurthy, W. C. Hasenplaugh, R. L. Allmon, and J. Enoch, "Variable precision floating point multiply-add circuit," *Intel Corporation*, Jul 2014.

[22] A. Ou, Q. Nguyen, Y. Lee, and K. Asanovi, "A case for mvps: Mixed-precision vector processors," *2nd International Workshop on Parallelism in Mobile Platforms (PRISM-2)*, Jun 2014.

[23] S. Mach, D. Rossi, G. Tagliavini, A. Marongiu, and L. Benini, "A transprecision floating-point architecture for energy-efficient embedded computing," in *International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, May 2018.

[24] C. Kim, J. Kim, S. Kim, D. Kim, N. Kim, G. Na, Y. H. Oh, H. G. Cho, and J. W. Lee, "Typed architectures: Architectural support for lightweight scripting," in *Proceedings 22nd International Conference on ASPLOS*, pp. 77–90, ACM, 2017.