Everything you Need to Know About Reduced Mixed Precision Computation in Numerical Programs

Dorra Ben Khalifa, Matthieu Martel

▶ To cite this version:

Dorra Ben Khalifa, Matthieu Martel. Everything you Need to Know About Reduced Mixed Precision Computation in Numerical Programs. 2023. hal-03978176

HAL Id: hal-03978176

https://hal.science/hal-03978176v1

Preprint submitted on 8 Feb 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NonCommercial - NoDerivatives 4.0 International License

Everything you Need to Know About Reduced Mixed Precision Computation in Numerical Programs

Dorra Ben Khalifa[1*] and Matthieu Martel[1,2*†]

[1*]LAMPS laboratory, University of Perpignan 52 Avenue Paul Alduy, Perpignan, 66100, , France.
[2]Bureaux du Polygone, Numalis, 256 Avenue Bureaux du Polygone, Montpellier, 34000, , France.

*Corresponding author(s). E-mail(s): dorra.ben-khalifa@univ-perp.fr; matthieu.martel@univ-perp.fr;
[†]These authors contributed equally to this work.

Abstract

In the recent decade, precision tuning becomes one of the key techniques to obtain significant gains in performance and energy efficiency. This process consists of substituting smaller data types to the original data types assigned to floating-point variables in numerical programs in such a way that accuracy requirements remain fulfilled. In face of the huge amount of precision tuning tools, we present in this article a survey on the approaches proposed in the bibliography. Besides, we point out the differences between these approaches and our own tool POP which avoids the usual trial-and-fail paradigm. POP implements a static analysis method relying on constraints derived from the modelling of the program errors propagation and accuracy specifications. We also extend the functionalities of POP by experimenting several optimization criteria to our system of constraints as a means to achieve the best performance improvements. We validate our results on a new set of numerical benchmarks and we report that in all cases we are able to accomplish better optimization by keeping the introduced numerical error below the given tolerance threshold.

Keywords: Precision optimization, computer arithmetic, performance metrics, error estimation.

1 Introduction

As science tries to answer bigger and deeper questions, data is growing faster than Moore's law [41], and the cost involved in developing new technology nodes is rising day after day. Not surprisingly, the increasing communication bandwidth entails more sophisticated protocols and higher speeds. Consequently, this will translate in increased power consumption up to the point where it becomes technically and economically unfeasible to further increase the communication bandwidth. For instance, the Top500[1] project report shows that supercomputer performance is approximately doubling every year, whilst power consumption is also rising. As of June 2020, the most powerful supercomputer Fugaku, number 1 of the Top500, is nowadays able to reach around 400 petaFLOPS (400×10^{15} FLOPS) in terms of computation power. This equates to assembling tens of millions of laptops together against the 148.6 petaFLOPS of the predecessor supercomputer, Summit.

To tackle this problem, designing reliable and energy efficient applications remains a real challenge to explore. In other words, designers seeking to reduce the energy usage should be helped in choosing the best implementations of their applications with regards to the targeted infrastructure. Consequently, energy-efficient heterogeneous supercomputers need to be coupled with software stacks able to exploit a range of techniques to trade-off between power, performance, and other metrics of quality to achieve the desired goals without exceeding the power envelope.

In these recent years, the precision tuning technique, which consists of using reduced precision number representations in numerical programs, has been widely recognized as one of the promising tools in the designer's arsenal. For instance, using FP32 single precision formats is often at least twice as fast as the FP64 double precision ones. By way of illustration, on AMD Opteron 246, IBM PowerPC 970, and Intel Xeon 5100, the single precision peak is twice the double precision peak [5]. By doing so, it can be possible to save memory and to have a positive impact on the footprint of programs concerning energy consumption, bandwidth usage and computation time.

In this context, various tools of precision tuning have been proposed recently to help developers select the most appropriate data representation. Such tools may integrate different approaches but their common goal is still to automatically or semi-automatically adapt an original code given in higher precision to the selected lower precision type. In our tool `POP` [2][2], no trial-and-fail method is employed. Instead, the accuracy of the arithmetic expressions assigned to variables is determined by semantic equations, in function of the accuracy of the operands. By reasoning on the number of significant bits of the variables of the program and by knowing the weight of their most significant bit thanks to a range analysis performed before the tuning phase, `POP` is able to reduce the problem to an Integer Linear Problem (ILP) which can be optimally solved in one shot by a classical linear programming solver. Concerning the number of variables, the method scales up to the solver limitations and the solutions are naturally found at bit level, making the number of data types considered for the tuning irrelevant. An important point is that the optimal solution to the continuous linear programming relaxation of our ILP is a vector of integers.

[1](https://www.top500.org/)
[2]Code source available at https://github.com/benkhelifadorra/POP-v2.0

By consequence, we may use a linear solver among real numbers whose complexity is polynomial [52] contrarily to the linear solvers among integers whose complexity is NP-Hard.

In this article, we focus on improving the efficiency of `POP` in several manners. We focus on experimenting different optimization criteria to the system of constraints generated by our tool in order to achieve the best compromise between performance and memory saving. Next, these optimization criteria are expressed as cost functions that the solver has to optimize. The first criteria optimizes only the sum of the accuracy of the variables assigned in the program. The second criteria is related to minimize the largest data type in the tuned programs. The third criteria corresponds to minimize the number of bits needed for each operation performed in the programs. The fourth optimization function aims to avoid type conversions of the same variables in the same program. We verify the effectiveness of these cost functions on a meaningful set of floating-point benchmarks coming from FPBench [19]. In all the benchmarks we considered, we show that were able to significantly reduce the amount of type cast operations and to limit the number of formats and the number of bits of operations, without significantly compromising the accuracy of the computation, which remains within a satisfactory threshold provided by the user.

The remainder of this article is as follows. We first review in Section 2 necessary background about finite-precision arithmetic which is an important building block for precision tuning, in particular, the IEEE754 Standard of floating-point arithmetic and the fixed-point arithmetic. An exhaustive survey about the precision tuning tools is given in Section 3. Section 4 details the approach implemented in our tool of precision tuning and introduces its new functionalities. An experimental evaluation of our extended approach is provided in Section 5. Finally, in Section 6 we draw some conclusions and propose future research directions.

2 Background on Computer Arithmetic

There exists several representations for approximating real numbers. Out of these representations, we will focus in this section on the floating-point (FP) arithmetic and fixed-point arithmetic.

2.1 Floating-Point Arithmetic

Currently, FP numbers are the most common representation used in numeric applications. Therefore, optimizing the use of FP formats is often a key to obtain high performance. In the following, we will focus on these formats, and in particular on the IEEE754 ones.

2.1.1 Normalized IEEE754 Binary Floating-Point

The IEEE754 Standard [4]is a technical standard for FP computation created by the Institute of Electrical and Electronic Engineers (IEEE). The standard formalizes a binary FP number x in base β, generally $\beta = 2$, as a triplet made of a sign, a mantissa and an exponent as shown in Equation (1), where $s \in \{\text{-}1,1\}$ is the sign, m represents

Format	Name	p	e bits	e_{min}	e_{max}
FP16	Half precision	11	5	−14	+15
FP32	Single precision	24	8	−126	+127
FP64	Double precision	53	11	−1122	+1223
FP128	Quadruple precision	113	15	−16382	+16383
FP256	Octuple precision	237	19	−262142	+262143

Table 1: Parameters defining the IEEE754 floating-point formats.

the mantissa, $m = d_0.d_1...d_{p-1}$, with the digits $0 \leq d_i < \beta$, $0 \leq i \leq p-1$, p is the precision (length of the mantissa) and the exponent $e \in [e_{min}, e_{max}]$.

$$x = s.m.\beta^{e-p+1} \tag{1}$$

We refer the reader to the Handbook of Floating-Point Arithmetic by Muller et al. [43] for a detailed and formal reference on the general subject of FP arithmetic.

2.1.2 Round-off Errors

Any result of a finite precision computation is subject to rounding errors. Consequently, this result that appears to be reasonable may therefore contain errors, and it may be difficult to judge how large the error is. For instance, if we have to compute the following operations on a typical calculator. First, $x = \sqrt{2}$, then $y = x^2$ and finally $z = y - 2$, i.e, the result should be $z = (\sqrt{2})^2 - 2$, which obviously is 0. The result reported by the calculator is $z = -1.38032020120975 \times 10^{-16}$. Let us note that when operands of arithmetic operations have themselves been subject to previous rounding, catastrophic loss of significant digits may happen and consequently the result may be completely false. While these errors are individually small, they propagate through a computation and can make its results meaningless [1].

However, one limitation of the FP arithmetic is that it requires dedicated support, either in hardware or in software, and depending on the application, this support may be too costly. Thus, the fixed-point arithmetic is an alternative which can be implemented with integers only.

2.2 Fixed-Point Arithmetic

A fixed number of digits is assigned to the sign, integer and fractional parts of the number within the data type format. As integer data types can be signed or unsigned, the sign field can be omitted also in fixed-point numbers. This is the case of unsigned fixed-point numbers, which represent the absolute value of the real number defined in Equation (2). Note that the binary point in fixed-point representation and the number of bits of each part are fixed. Thus, the scale factor of the associated data is constant and the range of the values that can be represented does not change during the computation.

$$(-1)^{sign} \times integer \cdot fractional \tag{2}$$

Let us also mention that many implementations of the fixed-point arithmetic use a two's complement representation instead of Equation (2). To be brief, Figure 1 presents the general representation of a number in fixed-point format composed of a

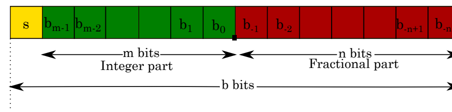

Fig. 1: Fixed-point representation of a signed number.

sign bit s (the most significant bit) and $b-1$ bits divided between the integer and the fractional parts. m and n represent the position of the radix point respectively to the most significant bit (*MSB*) and to the least significant bit (*LSB*).

Note that the recent version of `POP` [10] is able to apply the conversion from floating-point to fixed-point types by finding directly the minimal number of bits needed at each control point to get a certain accuracy on the results. Consequently, applying our method directly in hardware implementation on FGPAs and for implementation on processors with no floating-point hardware units is a very feasible task
.

3 State-of-the-Art: Tools for Precision Tuning

In recent years, precision tuning is emerging as a new trend to save the resources on the available processors, especially when new error-tolerant applications are considered [13].For example, many applications can tolerate some loss in quality during computation, as in the case of media processing, data mining, machine learning, etc. In addition, as we have showed in Section 2, almost all numerical computations are performed using floating-point operations to represent real numbers, the precision of the related data types should be adapted in order to guarantee the desired overall rounding error and to strengthen the performance of programs.

Consequently, designers should be helped to obtain the best trade-off between precision and performance by allocating some program variables in low precision (e.g. FP16 and FP32) and by using high precision (e.g. FP64 and FP128) selectively. For this reason, various tools have been proposed to help developers select the most appropriate data representations. Such tools may integrate different approaches but their common goal is still to automatically or semi-automatically adapt an original code given in higher precision to the selected lower precision type. The purpose of the present section is to review the existing literature on techniques and tools concerning precision tuning. What makes our review original is that we go further by incorporating the most recent tools and examining the strengths and shortcomings of each tool in comparison to our tool `POP`. After taking a closer look on the behaviour of each tool, we deduce two main classifications: *static* and *dynamic* analysis tools.

3.1 Static Analysis Tools

The main insight of the tools of this category is that they are able to extract additional knowledge from the program source code without executing it with input data. `Rosa` [20] is a source-to-source compiler which takes as input a real-valued program with error specifications and synthesizes code over an appropriate floating-point (FP32, FP64, FP128, and an extended format with 256 bit width) or fixed-point data

type (8, 16, 32 bit) which fulfills the specification. Rosa operates on a subset of the Scala programming language. In particular, the programmer writes the program in a real-valued specification language and makes numerical errors explicit in pre- and post-conditions. It is then up to Rosa to determine an appropriate data type which fulfills the specification and to generate the corresponding code. In addition, Rosa internally exploit the Z3 SMT solver [42] to process the precision constraints derived from the program accuracy specifications. However, Rosa handles conditional statements soundly by assigning only uniform precision to the variables of their programs.

Based on the *Symbolic Taylor Expansion* method, FPTuner [16] exposes a user-defined threshold for the amount of type casts that the tool may insert into the code. Let us state that the approach deployed by FPTuner is close to the one implemented in POP, especially in the constraint generation step. However, FPTuner relies on a local optimization procedure by solving quadratic problems for a given set of candidate data types. Contrarily to POP , FPTuner is limited to straight-line programs (without conditionals and loops). Unfortunately, it also requires a certain user skill for choosing which mixed-precision variants are more efficient and is thus not fully automated. Furthermore, its tuning time can be prohibitively large whereas we will show that the method embodied in POP is more fast and efficient even in the case of large codes.

In the context of precision tuning tool-chains, the TAFFO tool [15] is a LLVM-based tool-chain, which is packaged as a set of plugins for the Clang compiler. Its strategy is to collect statically annotations from the source code and it converts them into LLVM-IR metadata with the goal to replace floating-point operations with fixed-point operations to the extent possible. TAFFO is based on affine arithmetic [54]. This analysis is used to project on the output the error introduced by each fixed-point instruction. The advantage of TAFFO is that it supports both C and C++ programs and it can be provided as a plugin for LLVM which are feasible to be extended in POP in future work. In contrast to TAFFO, POP is able to return solutions at bit-level suitable for the IEEE754 floating-point arithmetic, the fixed-point arithmetic and the MPFR library for non-standard precision.

3.2 Dynamic Analysis Tools

A considerable share of precision tuning tools are based on dynamic analysis. The main insight of these techniques is to lower the precision of the values in the program and observe the error on the output of a testing run. Consequently, their majority apply a trial-and-fail paradigm to precision tuning. We distinguish tools which are based on search algorithms, tools oriented for GPU kernels and for neural networks.

3.2.1 Search Algorithm Based Tools

Precimonious [49] is a dynamic automated search-based tool that leverages the LLVM framework to tweak variable declarations to build and prototype mixed-precision configurations within a given error threshold. Precimonious is based on the delta-debugging algorithm search [56] which guarantees to find a local

1-minimum if one exists. A configuration is said to be 1-minimal if lowering any additional variable (or function call) leads to a configuration that produces an inaccurate result, or is not faster than the original program. Despite the fact that `Precimonious` can handle any program, including programs with loops, it presents several gaps. Unlike `POP` which optimizes all the variables of the program, `Precimonious` optimizes only the precision of declared variables. It uses external description files (JSON or XML) to declare which variables in the source code should be explored and which data types have to be investigated. Moreover, it estimates round-off errors by dynamically evaluating the program on several random inputs. By doing so, we can deduce that this approach is not sound because of the large number of program executions needed for a reasonably confident error bound. Comparing to `POP` which takes several seconds per benchmark, `Precimonious` uses dynamic evaluation to estimate the expected running time. However, this approach is not entirely reliable as running times can vary substantially between runs. Last of all, `Precimonious` does not use any knowledge on the structure of the program to identify potential variables of interest. This latter limitation has made the subject of several work which extended `Precimonious` in several manners.

`Blame Analysis` [50] is a dynamic technique which aims at reducing the space of variables of `Precimonious`. It performs shadow execution to identify variables that are numerically insensitive and which can consequently be excluded from the search space before tuning. The analysis finds a set of variables that can be in single precision, while the rest of the variables are in double precision. However, the output configurations may or may not improve performance, so to use the analysis in practice one must perform runs of the program to determine which configurations actually improve performance.

Another dynamic tool sharing some objectives and methodologies of `Precimonious` is called `PROMISE` [28]. It is written in Python and it relies on the `CADNA` software [32] to implement the *Discrete Stochastic Arithmetic* (DSA) verification in C, C++, and Fortran program source code. `PROMISE` automatically modifies the precision of variables taking into account an accuracy requirement on the computed result. Based on the delta-debugging search algorithm which reduces the search space of the possible variables to be converted, it provides a subset of the program variables which can be converted from FP64 to FP32 only. Meanwhile, `PROMISE` is able to tune programs only in FP32 single precision and it remains a time-intensive tool.

`HiFPTuner` [29] is another extension of the `Precimonious` tool which uses a hierarchical search approach. It combines a static analysis to create the hierarchical structure in order to minimize the number of type cast operations whereas the dynamic profiling highlights the hottest dependencies. A major limitation is that `HiFPtuner`'s configurations are dependent on the tuning inputs, and no accuracy guarantee is provided for untested inputs. Besides, It can be used to tune medium-sized programs only.

3.2.2 Similar Search-Based Tool

CRAFT [36] is a framework that performs an automated search of a program's instruction space, determining the level of precision necessary in the result of each instruction to pass a user-provided verification routine assuming all other operations are done in high precision such as FP64 double precision. CRAFT relies on the well-established Dyninst binary analysis toolkit to provide instrumented and mixed precision code. The original implementation of the CRAFT framework known as a binary mode version, relies on binary instrumentation and considers the whole program as its scope. Recently, its newer version is able to process the source code of the program and focus only on user-defined variables known as the variable mode version. Like Precimonious, CRAFT uses external description files (JSON or XML) to declare the variables and the data types to explore. While it uses heuristics to sample a fraction of the search space, it can be very time consuming even for very small programs. Finally, a tool called fpPrecisionTuning [30] performs a search over the mixed-precision search space using a user-given error bound, but this tool uses MPFR [26] and source code modification to simulate non-standard precision.

In summary, all the search-based approaches used for identifying valid mixed-precision configurations are time-intensive. In addition, their approaches does not scale with the number of possible number representations that can be used. Furthermore, discontinuities in the program can trick a greedy algorithm into a local optimum, which may be considerably distant from the global optimum.

3.2.3 GPU Applications Based Tools

Angerd et al. [3] have described a framework for precision tuning for GPU applications. They investigate an approximation of floating-point values in computer graphics kernels using three different low-precision formats: the IEEE754 formats (specifically FP16 and FP32), the mantissa truncation in which the data types are obtained by truncating mantissa bits from the basic IEEE754 formats, and finally a dynamically selected exponent and mantissa width, which are data types with variable bit width but constant ratio between the number of mantissa bits and that of exponent bits. Angerd et al. [3] extend LLVM with the custom defined data types and transparently converts the floating-point values. While these custom defined data types are not guaranteed to be supported by the target hardware, the proposed approach entails wrapping every memory access instruction to unpack and to pack the data from and to such data types. Consequently, this work is particularly pertinent for architectures where the cache and the memory size are critical, e.g. HPC accelerators. Although the main focus of this framework operates on hardware-heterogeneity-aware programming languages, such as OpenCL, they process the whole computational kernel and do not satisfy any user accuracy requirement on the output.

Not too far from this work concept, the work described by Nobre et al. [46] presents a LARA-based approach [12] for precision-tuning. It takes an OpenCL kernel as input and it generates and evaluates multiple versions of the input kernel. Those versions exploit mixed-precision data types to achieve performance improvements

over the original kernel, while they satisfy a user-defined constraint on the quality of the output. A similar work proposed by Rojek [48] presents a machine-learning based method for the dynamic selection of the precision level for GPU computation. It implements a modified version of the random forest algorithm to decide whether a variable type should be in FP32 or FP64 floating-point (no other data types are considered). Broadly speaking, the tools proposed by Nobre et al. [46] and by Rojek [48] operate on GPU kernels by using OpenCL/CUDA programming languages whereas Angerd et al. [3] propose a GPU-oriented approach while working within the intermediate representation of the compiler.

Other tools perform a static data flow analysis with a dynamic profiling on the source code. In this setting, `AMPT-GA` [33] is a tool oriented to GPU applications which combines static analysis for casting-aware performance modeling with dynamic analysis for enforcing precision constraints. Particularly, it performs a profile run of the application to identify the hottest computational kernels which may especially benefit from the precision reduction and a static analysis that only aims at identifying strongly connected variables in the dependency graph to attempt to assign the same data type to group of variables instead of acting on single variables.

`GPUMixer` [35] is one more tool designed for GPU kernels. This tool accepts CUDA kernels as input in the form of NVVM-IR intermediate representation, which can be generated by the Clang compiler front-end, and replaces the FP64 floating-point operations with FP32. It decides whether to apply the conversion to a code region or not depending on a tunable metric, which is based on the ratio between the number of affected arithmetic instructions and the number of type cast instructions. However, the tool is limited by the fact that the NVIDIA CUDA C programming guide does not specify the cost of all GPU operations.

`Autoscaler for C` [34] is a source-to-source compiler that complies with the ANSI C programming language. Its purpose is to convert every variable to fixed-point by using a data size which guarantees the absence of overflow. It performs an exploratory run over the original floating-point code to obtain an estimation of the dynamic range for each variable. Since the input and output of this translator are ANSI C compliant programs, it can be used for any fixed-point Digital Signal Processors (DSP) that supports ANSI C compiler. Another work not particularly different from the one implemented in [34] is described in [14]. The proposed method automates the floating-to-fixed point conversion by re-targeting the existing source-to-source compiler `GeCos` framework (Generic Compiler Suite), designed for use with hardware implementations, to produce code suitable for execution in HPC environments.

`ADAPT` [40] uses the reverse mode of algorithmic differentiation [45] to determine how much precision is needed in a program inputs and intermediate results in order to achieve a desired accuracy in its output, converting this information into precision recommendations. As the algorithmic differentiation approach views a computer program as a composition of a sequence of arithmetic operations, `ADAPT` uses this data to capture the propagation of errors through the data flow graph of the computation. It performs aggregation and analysis on this data along with

the original computation to determine the floating-point sensitivity of all the variables and operations in the program. Although the fact that ADAPT considers only IEEE754-compliant formats, it provides mixed-precision recommendations that satisfy a specified error threshold without requiring any search-based strategies. Later, ADAPT alongside CRAFT [36] and another tool called Typeforge have been incorporated in a framework named FloatSmith [38]. Broadly speaking, ADAPT provides the dynamic analysis of the code, CRAFT looks for the best precision tuning configuration, and Typeforge implements the source-to-source conversion for C/C++ code.

AMP [44] is a profile-driven tool that profiles applications to measure undesirable numerical behavior at the floating-point operation level. AMP takes a single precision application as input and its output is a mixed-precision application in which precision have been chosen to improve accuracy. The limitation of this tool is that it accepts only applications in which all operations are at the minimum precision. Otherwise, they should downgrade all the operations in higher precision (e.g. double precision) to single precision before applying their profiling. Indeed, although AMP monitor and locate numerical faults, it is infeasible to trace and quantify error propagation through every computational sequence of operations.

STOKE [51] is a general stochastic optimization and program synthesis tool to handle floating-point computation. Beginning from floating-point binaries produced either by a production compiler or written by hand, the tool shows that through repeated application of random transformations it is possible to produce high performance optimizations that are specialized both to the range of live-inputs of a code sequence and the desired precision of its live-outputs.

More recently, a tool called PyFloaT [11] has presented a methodology for tuning the precision of full fledged scientific applications written using multiple programming languages: Python, C++, CUDA and Fortran. It uses an instruction-centric analysis that uses call stack information and temporal locality to address the large scale of HPC scientific programs. We end up the list of the precision dynamic tools by the approach proposed by Yesil et al. [55] which is based on a proof concept for DPS (Dynamic Precision Scaling). The purpose of DPS is to run the program on reduced precision floating-point functional units whenever the data can tolerate the degradation, and to dynamically switch to the original floating-point data types when there is the need to preserve the accuracy. Moreover, we cite FloPoCo [24] an open source C++ framework written in C++ that generates VHDL code to design custom arithmetic data path of floating-point cores. Also, it generates a synthetizable hardware description according to the parameters specified via C++ code.

3.2.4 Precision Tuning for Neural Networks

Arnault et al. [31] introduced a static method for minimizing the precision in which the neurons of a neural network compute. Their method models the propagation of the round-off errors through a set of linear constraints among integers which can be solved by linear programming. Recently, the precision tuning of neural networks using fixed-point arithmetic has been studied in [9]. The fixed-point precision of each neuron is determined, taking into account a certain error threshold.

Tool	Input Language	Output Language	Data Types	Framework	Licence
ADAPT [40]	C/C++, Fortran	description	FP32, FP64	CodiPack	GNU GPL v3.0
AMPT-GA [33]	C/C++	description	IEEE754	LLVM	proprietary
Angerd et al. [3]	LLVM-IR	description	FP32, custom	LLVM 3.5	proprietary
AMP [53]	LLVM-IR	LLVM-IR	IEEE754	LLVM 3.4	proprietary
Autoscaler for C [34]	ANSI C	C++	fixed	SUIF	proprietary
CRAFT [36, 37]	x86 binary, C/C++	description, C/C++	FP32, FP64	Dyninst	GNU LGPL v3.0
FPTuner [16]	FPCore	FPCore	IEEE754	Gurobi 6.5	MIT
FloatSmith [38]	C/C++	description	FP32, FP64	CRAFT, ADAPT	GNU GPL v3.0
fpPrecisionTuning [30]	C	MPFR	IEEE754, fixed	C2mpfr	BSD, MIT
FloPoCo [24]	C++	VHDL	custom	-	proprietary
GPUMixer [35]	NVVM-IR	NVVM-IR	FP32, FP64	LLVM 4.0	proprietary
HiFPTuner [29]	LLVM-IR	description	FP32, FP64, FP128	Precimonious LLVM 3.8	BSD-3 Clause
Precimonious [49]	LLVM-IR	description	FP32, FP64, FP128	LLVM 3.0	BSD-3 Clause
PROMISE [28]	C/C++	C/C++	FP32 and FP64	CADNA for C/C++	GNU LGPL v3.0
PyFloaT [11]	Python, C++ CUDA, fortran	description	IEEE754	GOTCHA	MIT
Rojek [48]	CUDA	CUDA	FP32 and FP64	–	proprietary
Rosa [20]	Scala	Scala	IEEE754, fixed	Z3	BSD-2 Clause
STOKE [51]	x86 binary	fixed x86 – 64	IEEE754, fixed	JIT assembler	Apache2.0
TAFFO [15]	LLVM-IR	LLVM-IR	fixed	LLVM 8.0	MIT

Table 2: Precision tuning tools properties.

The work presented in [25] considers tuning the precision of an already trained neural network. Their methodology employs the PROMISE tool in order to obtain the lowest precision for each of its parameters, while keeping a certain accuracy on its results. The results obtained are compared for different neural networks.

3.2.5 Combining Tools

The automated tools of precision tuning are very often combined with other tools interested in error analysis and rewriting-based methods. In this section, we illustrate the benefits of composing complementary floating-point tools to achieve results neither tool provides in isolation. Our study reports combining tools for analysis and optimization and for rewriting and tuning.

Combining Tools for Analysis and Optimization

The first work that was interested in the combinations of tools is undoubtedly the Daisy tool [21]. It provides in a single tool the main building blocks for accuracy analysis of floating-point and fixed-point computations which have emerged from recent related work. In particular, Daisy extends the approach implemented in Rosa [20] by integrating the rewriting capabilities of Xfp [22]. Daisy integrates several techniques for sound analysis and optimization of finite-precision computations. Its is also able to provide a mixed precision solution that considers both floating-point and fixed-point data making it generally applicable to both scientific computing and embedded applications. Unlike our tool POP which is able to tune programs with expressions, loops, conditionals and even arrays, Daisy does not address conditional-based programs.

The authors of Daisy and Herbie have worked together to combine their tools in [6]. While Herbie optimizes the accuracy of straight-line floating-point expression, it employs a dynamic round-off error analysis and thus cannot provide sound guarantees on the results. Consequently, its combination with Daisy can help to check whether its unsound optimizations improved the worst-case round-off error or not. Meanwhile, this method do not handle loops and conditionals yet.

Combining Tools for Rewriting and Tuning

The main insight of rewriting techniques is to search through different evaluation orders to find one which minimizes the round-off error at no additional run-time cost. The mixed-precision tuning techniques aim to choose the smallest data type which still provides sufficient accuracy in order to save valuable resources like time, memory or energy. In this context, Anton is the first fully automated tool [23] that combines these two techniques in one single tool. The rewriting step is inspired from the xfp tool [22]. For the mixed-precision tuning step, Anton uses a variation of the delta-debugging algorithm used by Precimonious [49]. It starts with all variables in the highest available precision and attempts to lower variables in a systematic way until it finds that no further lowering is possible while still satisfying the given error bound. Although Anton tried to reduce its search space by using a static sound error analysis as well as a static performance cost function, the technique is limited to rather small programs that can be verified statically.

Another study to find an efficient precision with a better accuracy of variables of programs was presented in [18]. It consists of the former work of Martel [39] combined with the Salsa optimizing tool [17]. The principle of this study is to apply the forward and backward error analysis by abstract interpretation approach [39] to compute the least floating-point formats on the benchmarks of Salsa. Similarly to the Anton [23] tool, their rewriting technique is performed before the mixed-precision tuning.

The Pherbie also performs precision tuning and rewriting at the same time. Also, it adapts and extends techniques from the Herbie tool to automatically generate a set of candidate implementations, and derive a Pareto-optimal accuracy versus speed trade-off, for a given floating-point expression. Pherbie implements precision tuning by introducing rewrites that cast candidate sub-expressions to different

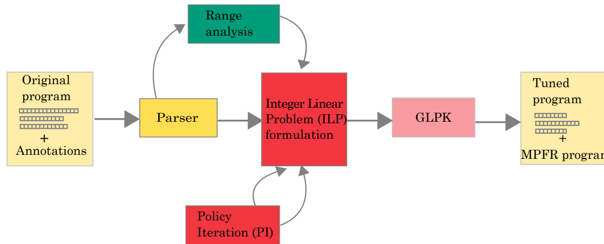

Fig. 2: Overall architecture of POP.

precision. Unlike our work, the analysis time of this tool can be exponential in the case of large programs containing a lot of expressions to rewrite which results in many new candidate implementations to manage and many calls to the Herbie tool.

The most challenging aspects of the precision tuning tools described in this section are outlined in Table 2. In particular, we report the input and output language of each tool, the supported formats considered by each tool, the framework and the licence.

To summarize, existing methods for precision tuning suffer from several limitations. The major drawback of the static analysis tools is fundamentally their incapacity to tune large codes with conditionals and loops. Nevertheless, a large amount of the dynamic tools follow a trial-and-fail strategy by reducing the precision of arbitrary chosen variables and executing or analyzing statically the program to see the new accuracy. In the next section, we introduce our tool POP and we demonstrate its new functionalities .

4 POP: A Precision Tuner Based on Formal Methods

4.1 POP in a Nutshell

Figure 2 summarizes the basic analysis steps of our tool POP [2, 8]. First, POP parses the input program and generates its equivalent syntactic tree thanks to the ANTLR parser generator v4.7.1[47]. While POP achieves only precision tuning, it uses a dynamic analysis which produces an under-approximation of the ranges of the variables for inputs taken randomly in user defined ranges. More precisely, what we use in the tuning is the unit in the first place of the values of the program defined hereafter in Equation (3).

$$\mathsf{ufp}(x) = \begin{cases} \min\{i \in \mathbb{Z} : 2^{i+1} > |x|\} = \lfloor \log_2(|x|) \rfloor & \text{if } x \neq 0 \\ e_{min} & \text{if } x = 0 \end{cases} \quad (3)$$

Some post-conditions added to the source code, e.g. the statement *require_nsb*$(x, 20)$, inform POP that the user wants to get on variable x an amount of 20 significant bits. In order to propagate the user information, POP will generate semantic equations modelling the propagation of the errors throughout the program source. The unknown variable to compute is the minimal number of significant bits needed for the input and intermediate variables of the program. This integer quantity is denoted by nsb.

```
1   a = 1.0;
2   i = 1.0;
3   x = 0.0;
4   while(i < 10.0){
5           a = a + 1.0;
6           x = x + a;
7           i = i + 1.0;
8   } ;
9    require_nsb(x,20);
```

Simple C program

```
1   a||19|| = 1.0||19||;
2   i = 1.0;
3   x||20|| = 0.0||20||;
4   while (i <10.0){
5     a||20|| = a||19|| +||20|| 1.0||20||;
6     x||20|| = x||20|| +||20|| a||20||;
7     i = i + 1.0;
8   };
9   require_nsb(x,20);
```

Program annotated with precision

Fig. 3: A straightforward C program before and after POP analysis.

Formally, if we consider that $\hat{x}$ is the approximation of a real number x in finite precision and if we have $\varepsilon(x) = |x - \hat{x}|$ be the absolute error, then we have

$$\varepsilon(x) \leq 2^{\mathsf{ufp}(x)-k+1} \quad . \tag{4}$$

Once the semantic equations are generated, POPcalls the GLPK linear solver. Finally, POP generates the optimized program annotated with the new number of significant bit nsb for each variable in the program with respect to accuracy desired by the user. A second method that optimizes the previous ILP formulation is also implemented in POP for In this method, we go one step further by introducing a second set of semantic equations. These new equations make it possible to tune even more the precision by being less pessimistic on the propagation of carries in arithmetic operations. However, the problem does not reduce any longer to an ILP problem (min and max operators are needed). Then we use policy iteration method to find efficiently the solution. In this article, we omit details about this second method and we only consider the pure ILP formulation.

Let us consider the simple C program of Figure 3. In this example, we suppose that all variables are in double precision before analysis (FP64). The original program is depicted in the top part of Figure 3. Some points can be highlighted about this example. For instance, we have the statement **require_nsb(x,20)** (Line 9) which informs the tool that the user wants to get on variable x only 20 significant bits. After analysis, we obtain in the bottom part of Figure 3 the minimal precision needed for the inputs and intermediary results satisfying the user assertion.

Since, in this example, 20 bits only are required for x, the result of the addition x$+$ a also needs 20 accurate bits only as shown in the bottom part of Figure 3 (precision are given in blue). If we want these precision in the IEEE754 mode, the nsb obtained

at bit-level is approximated by the upper number of bits corresponding to a IEEE754 format. For example, variable at Line 6 we have $\mathsf{nsb}(x) = 20$ bits, then `x` is tuned to the FP32 single precision. Also, our recent tool `POPiX` [10] transforms a given numerical floating-point program into semantically equivalent one that exploits fixed-point computations with integers only[3].

4.2 Integer Linear Programming with Multiple Objective Functions

To obtain the optimal solution to our system of constraints, cost functions are given to the linear solver as optimization objective functions [7]. Depending on which cost function is used by `POP`, different criteria may be considered for the tuning. Below we propose four optimization criteria. The first criterion is the default cost function used by our tool. It consists in minimizing the sum of the nsb quantities of all the variables assigned in the program. The remaining functions constitute the new extension of `POP`. They are related to the largest data type, the number of bits needed for each operation and the prohibition of type conversions. We underline the fact that since `POP` is based on a system of constraints, assigning to it new optimization objectives can be done easily, without a deep refactoring of the tool.

4.3 Minimize the Sum of Number of Significant Bits of the Variables

The default cost function in `POP` is to minimize the sum of the precision of the assigned variables in the program. More formally, let *Lab* denotes the set of labels of the program and let $T : Lab \to \mathbb{N}$ be a tuning assigning to each control point $\ell \in Lab$ an integer precision. We denote $\mathscr{T}$ the set of correct tuning. This cost function, denoted by $F0$, is given as shown in Equation (5).

$$F0 = \min_{T \in \mathscr{T}} \left\{ \sum_{\ell \in Lab} T(\ell) \right\} \tag{5}$$

However, this cost function may lead to cases where some variables have large formats and others small ones (e.g. from FP16 half precision to FP64 double) which makes difficult hardware optimizations.

4.4 Minimize the Number of the Largest Data Type

The purpose of this cost function is to find the minimal number of bits of the greatest format needed in the program. For instance, the question that we may answer with this technique is the following: *if the user wants to obtain a result with only 18 significant bits, will all variables be defined as single precision numbers (FP32)?* Consequently, this cost function is very useful when using processors with limited formats. We denote by $F1$ The cost function for maximal precision that we aim to compute as shown in Equation (6).

$$F1 = \min_{T \in \mathscr{T}} \left\{ \max_{\ell \in Lab} T(\ell) \right\} \tag{6}$$

[3] https://github.com/sbessai/popix

4.5 Minimize of the Operations Number of Bits

Our third cost function $F2$ focuses on the operators instead of the variables of the program. We aim at minimizing only the number of bits used in the arithmetic operations, without considering what is used for variables. The interest is to minimize the hardware needed to run the programs. Also, this optimization is particularly relevant for circuit implementations, e.g. using FPGAs [27]. Formally, let $Op \subseteq Lab$ be the subset of labels attached to operators such as additions, multiplications, elementary functions, etc. Here, we aim at computing

$$F2 = \min_{T \in \mathscr{T}} \left\{ \sum_{\ell \in Op} T(\ell) \right\} \quad . \tag{7}$$

In the present work, we assign the same weight to each operation (i.e. its number of bits). However, it would be interesting to assign different weights, for instance to take into account that a multiplication is more costly than an addition at the hardware level (same for elementary functions.)

4.6 Minimize type conversions for the Occurrence of the Variable

Mixed precision tuning, as done by `POP`, offers the advantage of optimizing the precision of a variable at each of its occurrences. However, from a performance point of view, this introduces type conversions which may slow down the programs. Let $\mathscr{V} : Var \to \text{wp}(Lab)$ be a function mapping each variable x of a program to the set of labels corresponding to the occurrences of x and let $Dom(\mathscr{V})$ denote the definition domain of $\mathscr{V}$. We add this new cost function in `POP`, denoted by $F3$, which enforces it to produce an uniform tuning by adding the constraints

$$\forall x \in Dom(\mathscr{V}),\ \forall \ell_1,\ \ell_2 \in \mathscr{V}(x),\ T(\ell_1) = T(\ell_2) \quad . \tag{8}$$

Let us remark that, in this mode, POP still achieve bit-level precision tuning. However this tuning is uniform and only one precision is returned for each variable which avoids type conversions.

5 Experimental Evaluation

The main goal of this experimental evaluation is to answer the following research questions:

*RQ*1. Which cost function is more efficient in terms of precision tuning optimization?
*RQ*2. Which cost function can give us the optimal type configurations of the variables in floating-point arithmetic?
*RQ*3. What is the impact of the precision loss parameter of the trigonometric functions on the `POP` tuning results for the different cost functions used for our benchmarks?
*RQ*4. Which cost function achieves the best performance improvements in terms of measured relative error and analysis time?

5.1 Experimental Setup

We evaluate `POP` on a new set of applications coming from FPBench, a synthetic benchmark for floating-point performance. These benchmarks are coming from

		F0			F1			F2			F3		
Benchmarks	**call**	8_{bits}	16_{bits}	32_{bits}	8_{bits}	16_{bits}	32_{bits}	8_{bits}	16_{bits}	32_{bits}	8_{bits}	16_{bits}	32_{bits}
azimuth	7	77	62	31	50	50	✗	77	62	31	14	12	9
carbonGas	0	84	71	✗	45	45	✗	84	71	✗	28	27	✗
CRadius	1	81	66	36	45	45	✗	81	66	36	28	23	13
CTheta	1	82	67	37	45	45	✗	81	66	36	28	23	13
doppler1	0	86	71	41	45	45	✗	86	71	41	23	21	18
doppler2	0	88	73	42	45	45	✗	88	73	42	23	22	19
doppler3	0	86	71	41	45	45	✗	86	71	41	22	21	18
instantCurrent	3	84	72	42	47	47	✗	83	70	40	19	19	18
jetEngine	0	79	64	33	51	51	✗	79	64	33	13	13	12
LowPassFilter	0	81	68	40	49	49	✗	96	96	96	19	19	19
CX	1	80	65	35	46	46	✗	80	65	35	14	12	8
CY	1	80	65	35	46	46	✗	80	65	35	14	12	8
triangle12	1	77	62	32	49	49	✗	77	62	32	17	15	10
turbine1	0	84	70	41	46	46	✗	84	70	41	17	16	13
turbine2	0	84	69	38	45	45	✗	84	69	38	5	5	5
turbine3	0	85	71	44	46	46	✗	85	71	44	17	17	15

Table 3: Percentage of `POP` optimization in 4, 8 and 16 bits for the synthesized program for the different cost functions.

different domains such as: mathematical libraries, Internet of Things, embedded systems, etc. Each program is evaluated with three accuracy requirements arbitrarily chosen by the user: 4, 8 and 16 bits which bound the relative error of the result. We use the optimization criteria F_0, F_1, F_2 and F_3 that we have already defined in equations (5), (6), (7) and (8) respectively, (see Section 4).

We run all our experiments on an Intel Core i5-8350U CPU cadenced at 1.7GHz on a Linux machine with 8 GB RAM.

5.2 Results Analysis

Table 3 shows the percentage of optimization given for each cost function tested by `POP` for all the user accuracy requirements. The first left-most column headed "call" refers to the number of elementary functions in the code. This information is very useful if we want to show the impact of manipulating the loss of precision parameter in functions as we have done in Figure 4. The ✗ symbol denotes that no feasible solution was returned by the solver for a given user accuracy. We assume that 100% is the percentage of all variables initially in FP64 double precision. The first observation is that with functions $F0$ and $F2$ we obtain almost the same optimization results for all the benchmarks in 8, 16 and 32 bits. Also, it is clear that the percentage of optimization decreases as the user accuracy requirements increase. For instance in "doppler1", 88% is the percentage of optimization with $F0$ for 8 bits of requirements against only 41% for 32 bits. Our second observation concerns $F1$. For 32 bits of requirements, `POP` is not able to tune the programs with this cost function. What explains this result is that for this given accuracy of 32 bits, minimizing the largest type of data is equivalent to tuning all variables in double precision which is the

		8 bits				**16 bits**				**32 bits**			
Benchmarks	**cost**	**H**	**S**	**D**	**%**	**H**	**S**	**D**	**%**	**H**	**S**	**D**	**%**
azimuth	F0	11	7	0	69%	0	18	0	54%	0	0	18	0%
	F1	0	3	15	9%	0	3	15	9%	0	11	7	33%
	F2	11	7	0	69%	0	18	0	54%	✗	✗	✗	✗
	F3	2	0	16	8%	0	2	16	6%	0	0	18	0%
carbonGas	F0	12	0	0	82%	1	12	0	59%	✗	✗	✗	✗
	F1	0	0	14	0%	0	0	14	0%	✗	✗	✗	✗
	F2	12	0	0	82%	1	12	0	59%	✗	✗	✗	✗
	F3	2	1	11	15%	1	2	11	13%	✗	✗	✗	✗
CRadius	F0	6	0	0	79%	0	6	0	54%	0	0	6	0%
	F1	0	0	6	0%	0	0	6	0%	✗	✗	✗	✗
	F2	6	0	0	79%	0	6	0	54%	0	0	6	0%
	F3	2	0	4	26%	0	2	4	18%	0	0	6	0%
doppler1	F0	10	0	0	79%	0	10	0	54%	0	0	10	0%
	F1	0	0	10	0%	0	0	10	0%	✗	✗	✗	✗
	F2	10	0	0	79%	0	10	0	54%	0	0	10	0%
	F3	1	0	9	7%	0	1	9	5%	0	0	10	0%
instantCurrent	F0	16	9	0	73%	8	19	0	63%	0	9	19	17%
	F1	0	0	28	0%	0	0	28	0%	✗	✗	✗	✗
	F2	16	9	0	73%	6	21	0	61%	0	7	21	13%
	F3	1	2	25	6%	0	3	25	5%	0	2	26	3%
jetEngine	F0	17	14	0	68%	0	26	5	45%	0	0	31	0%
	F1	0	11	20	19%	0	11	20	19%	✗	✗	✗	✗
	F2	17	14	0	68%	0	26	5	45%	0	0	31	0%
	F3	1	0	30	2%	0	1	30	1%	0	0	31	0%
lowPassFilter	F0	18	12	0	78%	0	20	0	58%	0	0	20	0%
	F1	0	0	22	0%	0	0	22	0%	✗	✗	✗	✗
	F2	18	2	0	78%	0	20	0	58%	0	0	20	0%
	F3	0	0	20	0%	0	0	20	0%	0	0	20	0%
CX	F0	7	0	0	79%	0	7	0	54%	0	0	7	0%
	F1	0	0	7	0%	0	0	7	0%	✗	✗	✗	✗
	F2	7	0	0	79%	0	7	0	54%	0	0	7	0%
	F3	1	0	6	11%	0	1	6	7%	0	0	7	0%
triangle12	F0	7	6	0	67%	0	13	0	54%	0	0	13	%
	F1	0	3	10	12%	0	3	10	12%	✗	✗	✗	✗
	F2	7	6	0	67%	0	13	0	54%	0	0	13	0%
	F3	2	0	11	12%	0	2	11	8%	0	0	13	0%
turbine1	F0	18	0	0	80%	0	18	0	57%	0	0	18	0%
	F1	0	0	19	0%	0	0	19	0%	✗	✗	✗	✗
	F2	18	0	0	80%	0	18	0	57%	0	0	18	0%
	F3	2	0	16	13%	0	2	16	11%	0	0	18	0%

Table 4: Analysis results and performance. The column ”cost” gives the cost function activated in `POP` For each selected user accuracy requirement (8, 16 and 32 bits), we give the type configuration found in half precision ”H”, single precision ”S” and double precision ”D” and the percentage of optimization ”%”.

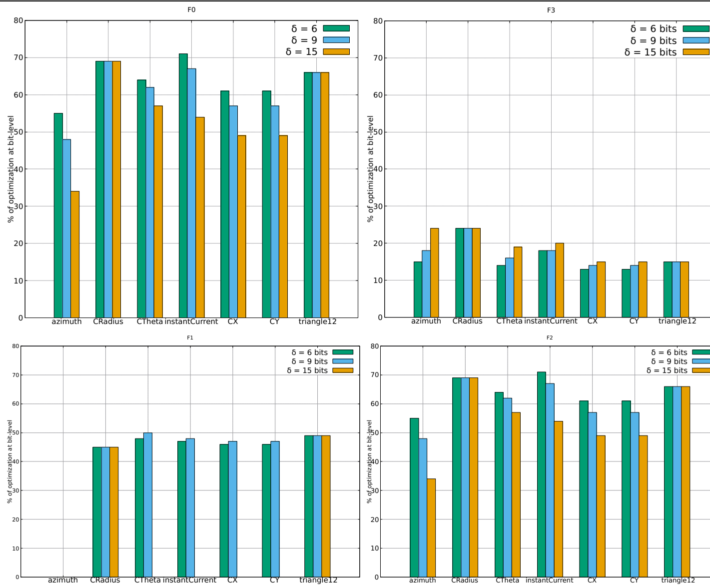

Fig. 4: Percentage of optimization at bit-level measured by varying the parameter of loss of precision φ of the elementary functions during the analysis.

initial case for these programs. Finally, by interpreting the results obtained with the cost function $F3$, we can deduce that the optimization rate is not as important as that of the other functions. This means that our programs analyzed do not contain many occurrences of the same variable.

Table 4 translates the precision found by `POP` in bit-level into the IEEE754 formats. The key point is to approximate the precision obtained by the upper number of bits corresponding to a IEEE754 format. For instance, in the analysis of the "carbonGas" program with $F0$ and with 8 bits requirement by the user, we obtain that 12% are converted from FP64 double precision (initially) into Half precision giving a total of 82% of overall optimization.

The elementary functions such as the natural logarithm, the exponential functions and the hyperbolic and trigonometric functions are not included in any arithmetic Standard when compared to the square root function which is included in the IEEE754 Standard. For this reason, each implementation of these functions has its own accuracy which we have to know. The purpose of the experimentation showed in Figure 4 is to consider that each elementary function introduces a loss of precision of φ bits, where $\varphi \in \mathbb{N}$ is a parameter of the analysis. We notice in Figure 4 that we only evaluate programs that contain elementary functions (identified thanks to the "call" column in Table 3). Let us note that in this experimentation we have fixed the user accuracy requirement to 16 bits. By varying the parameter $\varphi = 6$, 9 and 15 bits, we get different results for each of our cost functions. In the top left side of Figure 4, we observe that for $\varphi = 6$ bits, we obtain a better optimization for the majority of the programs reaching up to 72% for the "instantCurrent" program. This

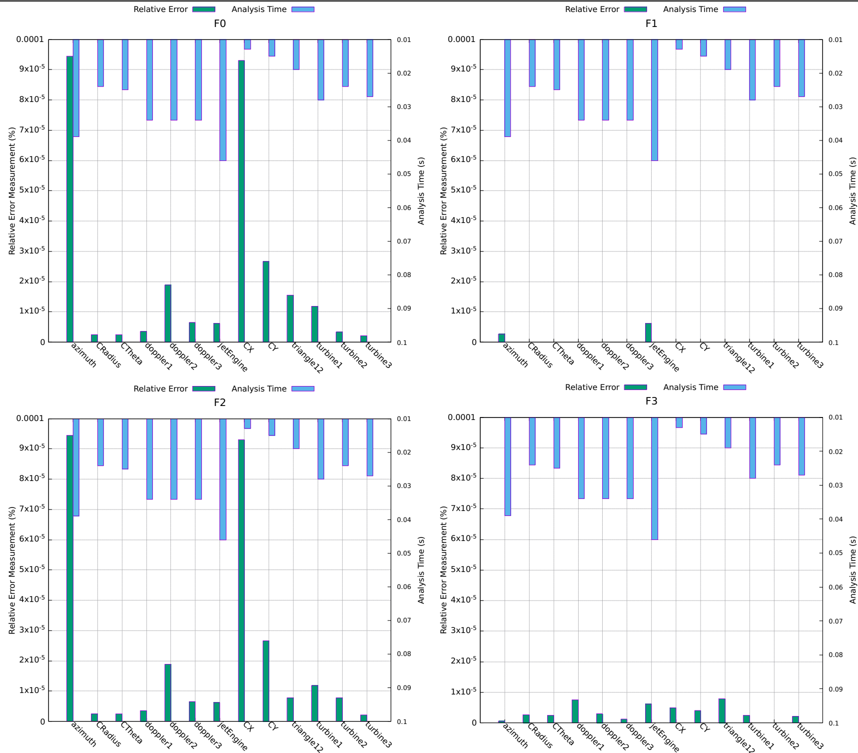

Fig. 5: Measured relative error and analysis time with respect to the activated cost function and the user defined accuracy.

observation changes for the $F3$ cost function when the percentage of optimization does not exceed 25% while the parameter $\varphi = 15$ bits give the best optimization. For the cost function $F1$, no solution was found for the "azimuth" program for the three parameters. Also, for $\varphi = 15$ bits, there is no optimization for the majority of the benchmarks with the $F1$ function. for the remaining cost function $F2$, the behavior of POP is similar to the results obtained for $F0$. The measured relative errors and analysis time are reported in Figure 5. We consider that a result has n significant bits if the relative error between the exact and approximated results is less than 2^{-n}. Noting that in this experiment, we assume that $n = 16$ bits (given by the user). The relative error is measured with respect to the original program where all variables are in double precision and the program returned by POP with the optimized precision. For the majority of the benchmarks and for all the cost functions tested, we can see that the relative error measure remains below the given user tolerance threshold. Also, we achieve best performance improvements when using the $F1$ and $F3$ optimization criteria. For instance, with $F1$ the errors measured are very small and can even be cancelled for some benchmarks. Concerning the time of analysis spent by POP, The histogram bars in the Figure 5 show that our method is very fast and the analysis time

remains negligible for analyzing our medium to large programs and for finding the new optimized formats. This is an advantage of POP over the other state-of-the-art tools that consume a lot of time and memory consuming.

6 Conclusion and Perspectives

In this article, we reviewed research work related to the precision tuning tools. Also, we have extended our tool POP with new optimization criteria in order to obtain trade-offs between, precision, analysis time and memory consumption. This extension have showed that several factors are important to optimize the accuracy of numerical programs such as the manipulation of optimization criteria and the implementation of elementary functions. We have evaluated our method on a new set of benchmarks. The results discussed show that for the majority of our benchmarks and with respect to the accuracy requirements given by the user, our tool succeeded in minimizing the number of significant bits of the variables, limiting the number of formats, the number of bits of operations and the number of type conversions between the variables. We shed the light that these results are helpful in the hardware level, especially for some processors that are limited to specific formats.

In future work, our efforts will focus on exploiting the fixed-point numerical representation by considering the fact that some architectures are more suited to fixed-point computations than others. The case studies for this point will belong to control applications. Also, we will generalize our technique to Deep Neural Networks for which it is important to save memory usage and computational resources.

7 Data Availability

The datasets generated during the current study are on publicly available on https://github.com/dbenkhal/POP-v2.0. The Fixed-point version of POP is available on https://github.com/sbessai/popix. However, the new functionnalities including the cost functions are available from the corresponding author on reasonable request.

References

[1] (1992) Patriot missile defense: Software problem led to system failure at dhahran, saudi arabia. Tech. Rep. GAO/IMTEC-92-26, General Accounting office

[2] Adjé A, Ben Khalifa D, Martel M (2021) Fast and efficient bit-level precision tuning. In: Static Analysis - 28th International Symposium, SAS 2021, Chicago, Illinois. Springer, Lecture Notes in Computer Science

[3] Angerd A, Sintorn E, Stenström P (2017) A framework for automated and controlled floating-point accuracy reduction in graphics applications on gpus. ACM Trans Archit Code Optim 14(4):46:1–46:25

[4] ANSI/IEEE (2008) IEEE Standard for Binary Floating-point Arithmetic. ANSI/IEEE, std 754-2008 edn.

[5] Baboulin M, Buttari A, Dongarra JJ, et al (2009) Accelerating scientific computations with mixed precision algorithms. Comput Phys Commun 180(12):2526–2533

[6] Becker H, Panchekha P, Darulova E, et al (2018) Combining tools for optimization and analysis of floating-point computations. In: Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings, Lecture Notes in Computer Science, vol 10951. Springer, pp 355–363

[7] Ben Khalifa D, Martel M (2022) Constrained precision tuning. In: 8th International Conference on Control, Decision and Information Technologies, CoDIT 2022, Istanbul, Turkey, May 17-20, 2022. IEEE, pp 230–236

[8] Ben Khalifa D, Martel M, Adjé A (2019) POP: A tuning assistant for mixed-precision floating-point computations. In: Formal Techniques for Safety-Critical Systems - 7th International Workshop, FTSCS 2019, Communications in Computer and Information Science, vol 1165. Springer, pp 77–94

[9] Benmaghnia H, Martel M, Seladji Y (2022) Code generation for neural networks based on fixed-point arithmetic. ACM Trans Embed Comput Syst Just Accepted

[10] Bessaï S, Ben Khalifa D, Benmaghnia H, et al (2022) Fixed-point code synthesis based on constraint generation. In: Design and Architecture for Signal and Image Processing - 15th International Workshop, DASIP 2022, Budapest, Hungary, June 20-22, 2022, Proceedings, Lecture Notes in Computer Science, vol 13425. Springer, pp 108–120

[11] Brunie H, Iancu C, Ibrahim KZ, et al (2020) Tuning floating-point precision using dynamic program information and temporal locality. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020. IEEE/ACM, p 50

[12] Cardoso JMP, Carvalho T, Coutinho JGF, et al (2012) LARA: an aspect-oriented programming language for embedded systems. In: Proceedings of the 11th International Conference on Aspect-oriented Software Development, AOSD 2012, Potsdam, Germany, March 25-30, 2012. ACM, pp 179–190

[13] Cherubin S, Agosta G (2020) Tools for reduced precision computation: A survey. ACM Comput Surv 53(2)

[14] Cherubin S, Agosta G, Lasri I, et al (2017) Implications of reduced-precision computations in HPC: performance, energy and error. In: Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing, ParCo 2017, 12-15 September 2017, Bologna, Italy, Advances in Parallel Computing, vol 32. IOS Press, pp 297–306

[15] Cherubin S, Cattaneo D, Chiari M, et al (2020) TAFFO: tuning assistant for floating to fixed point optimization. IEEE Embed Syst Lett 12(1):5–8

[16] Chiang W, Baranowski M, Briggs I, et al (2017) Rigorous floating-point mixed-precision tuning. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. ACM, pp 300–315

[17] Damouche N (2016) Improving the numerical accuracy of floating-point programs with automatic code transformation methods. (amélioration de la précision numérique de programmes basés sur l'arithmétique flottante par les méthodes de transformation automatique). PhD thesis, University of Perpignan, France

[18] Damouche N, Martel M (2018) Mixed precision tuning with salsa. In: Proceedings of the 8th International Joint Conference on Pervasive and Embedded Computing and Communication Systems, PECCS 2018, Porto, Portugal, July 29-30, 2018. SciTePress, pp 185–194

[19] Damouche N, Martel M, Panchekha P, et al (2016) Toward a standard benchmark format and suite for floating-point analysis. In: Numerical Software Verification - 9th International Workshop, NSV, Revised Selected Papers, pp 63–77

[20] Darulova E, Kuncak V (2017) Towards a compiler for reals. ACM Trans Program Lang Syst 39(2):8:1–8:28

[21] Darulova E, Volkova A (2019) Sound approximation of programs with elementary functions. In: Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II, Lecture Notes in Computer Science, vol 11562. Springer, pp 174–183

[22] Darulova E, Kuncak V, Majumdar R, et al (2013) Synthesis of fixed-point programs. In: Proceedings of the International Conference on Embedded Software, EMSOFT 2013, Montreal, QC, Canada, September 29 - Oct. 4, 2013. IEEE, pp 22:1–22:10

[23] Darulova E, Horn E, Sharma S (2018) Sound mixed-precision optimization with rewriting. In: Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems, ICCPS 2018, Porto, Portugal, April 11-13, 2018. IEEE Computer Society / ACM, pp 208–219

[24] de Dinechin F, Pasca B (2011) Designing custom arithmetic data paths with flopoco. IEEE Des Test Comput 28(4):18–27

[25] Ferro Q, Graillat S, Hilaire T, et al (2022) Neural network precision tuning using stochastic arithmetic. In: Software Verification and Formal Methods for ML-Enabled Autonomous Systems - 5th International Workshop, FoMLAS 2022, and 15th International Workshop, NSV 2022, Proceedings, Lecture Notes in Computer Science, vol 13466. Springer, pp 164–186

[26] Fousse L, Hanrot G, Lefèvre V, et al (2007) Mpfr: A multiple-precision binary floating-point library with correct rounding. ACM Trans Math Softw 33

[27] Gao X, Constantinides GA (2015) Numerical program optimization for high-level synthesis. In: Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, February 22-24, 2015. ACM, pp 210–213

[28] Graillat S, Jézéquel F, Picot R, et al (2019) Auto-tuning for floating-point precision with discrete stochastic arithmetic. J Comput Sci 36

[29] Guo H, Rubio-González C (2018) Exploiting community structure for floating-point precision tuning. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018. ACM, pp 333–343

[30] Ho N, Manogaran E, Wong W, et al (2017) Efficient floating point precision tuning for approximate computing. In: 22nd Asia and South Pacific Design Automation Conference, ASP-DAC 2017, Chiba, Japan, January 16-19, 2017. IEEE, pp 63–68

[31] Ioualalen A, Martel M (2019) Neural network precision tuning. In: Quantitative Evaluation of Systems, 16th International Conference, QEST 2019, Glasgow, UK, September 10-12, 2019, Proceedings, Lecture Notes in Computer Science, vol 11785. Springer, pp 129–143

[32] Jézéquel F, Chesneaux JM (2008) CADNA: a library for estimating round-off error propagation. Comput Phys Commun 178(12):933–955

[33] Kotipalli PV, Singh R, Wood P, et al (2019) AMPT-GA: automatic mixed precision floating point tuning for GPU applications. In: Proceedings of the ACM International Conference on Supercomputing, ICS. ACM, pp 160–170

[34] Kum KI, Kang J, Sung W (2000) Autoscaler for c: an optimizing floating-point to integer c program converter for fixed-point digital signal processors. IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing 47(9):840–848

[35] Laguna I, Wood PC, Singh R, et al (2019) Gpumixer: Performance-driven floating-point tuning for GPU scientific applications. In: High Performance Computing - 34th International Conference, ISC High Performance 2019, Frankfurt/Main, Germany, June 16-20, 2019, Proceedings, Lecture Notes in Computer Science, vol 11501. Springer, pp 227–246

[36] Lam MO, Hollingsworth JK, Stewart GW (2013) Dynamic floating-point cancellation detection. Parallel Comput 39(3):146–155

[37] Lam MO, Hollingsworth JK, de Supinski BR, et al (2013) Automatically adapting programs for mixed-precision floating-point computation. In: International Conference on Supercomputing, ICS'13, Eugene, OR, USA - June 10 - 14, 2013. ACM, pp 369–378

[38] Lam MO, Vanderbruggen T, Menon H, et al (2019) Tool integration for source-level mixed precision. In: 2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness), Denver, CO, USA, November 18, 2019. IEEE, pp 27–35

[39] Martel M (2017) Floating-point format inference in mixed-precision. In: NASA Formal Methods - 9th International Symposium, NFM, pp 230–246

[40] Menon H, Lam MO, Osei-Kuffuor D, et al (2018) ADAPT: algorithmic differentiation applied to floating-point precision tuning. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018. IEEE / ACM, pp 48:1–48:13

[41] Moore GE (1965) Cramming more components onto integrated circuits. Electronics 38(8)

[42] de Moura LM, Bjørner N (2008) Z3: an efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, LNCS, vol 4963. Springer, pp 337–340

[43] Muller J, Brisebarre N, de Dinechin F, et al (2010) Handbook of Floating-Point Arithmetic. Birkhäuser

[44] Nathan R, Anthonio B, Lu S, et al (2014) Recycled error bits: Energy-efficient architectural support for floating point accuracy. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014. IEEE Computer Society, pp 117–127

[45] Naumann U (2012) The Art of Differentiating Computer Programs - An Introduction to Algorithmic Differentiation, Software, environments, tools, vol 24. SIAM

[46] Nobre R, Reis L, Bispo J, et al (2018) Aspect-driven mixed-precision tuning targeting gpus. In: Proceedings of the 9th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and 7th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms, PARMA-DITAM@HiPEAC 2018, Manchester, United Kingdom, January 23-23, 2018. ACM, pp 26–31

[47] Parr T (2013) The Definitive ANTLR 4 Reference, 2nd edn. Pragmatic Bookshelf

[48] Rojek K (2019) Machine learning method for energy reduction by utilizing dynamic mixed precision on gpu-based supercomputers. Concurr Comput Pract Exp 31(6)

[49] Rubio-González C, Nguyen C, Nguyen HD, et al (2013) Precimonious: tuning assistant for floating-point precision. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13. ACM, pp 27:1–27:12

[50] Rubio-González C, Nguyen C, Mehne B, et al (2016) Floating-point precision tuning using blame analysis. In: Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016. ACM, pp 1074–1085

[51] Schkufza E, Sharma R, Aiken A (2014) Stochastic optimization of floating-point programs with tunable precision. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014. ACM, pp 53–64

[52] Schrijver A (1998) Theory of linear and integer programming. John Wiley & Sons

[53] Seetharam K, Keh LOCT, Nathan R, et al (2013) Applying reduced precision arithmetic to detect errors in floating point multiplication. In: IEEE 19th Pacific Rim International Symposium on Dependable Computing, PRDC 2013, Vancouver, BC, Canada, December 2-4, 2013. IEEE Computer Society, pp 232–235

[54] Stolfi J, Figueiredo LHD (2003) An introduction to affine arithmetic. Trends in Applied and Computational Mathematics 4:297–312

[55] Yesil S, Akturk I, Karpuzcu UR (2018) Toward dynamic precision scaling. IEEE Micro 38(4):30–39

[56] Zeller A, Hildebrandt R (2002) Simplifying and isolating failure-inducing input. IEEE Trans Softw Eng 28(2):183–200