Using loop transformations for precision tuning in iterative programs

Youssef Fakhreddine and Guillaume Revy
Univ Perpignan Via Domitia, DALI, Perpignan, France
LIRMM, Univ Montpellier, CNRS (UMR 5506), Montpellier, France

***Abstract*—Many floating-point formats are available, providing all different levels of precision. By mixing several of these formats in the same program, it is possible to achieve good performance while maintaining an acceptable level of output accuracy. Therefore various tools have been designed to adapt the precision of computations in floating-point programs for performance and accuracy purposes. However most of them do not consider the iterative nature of these programs. This article presents a tool that enables to adapt the precision of floating-point computations in iterative routines, at the iteration level. This tool is based on multiple-precision computations to evaluate the impact of some format adaptations on the output accuracy, and it uses the delta-debugging to isolate the most relevant instruction set to be tuned. The originality of our approach is that it relies on static loop transformations to duplicate loop body instructions, and thus to increase the number of possible instructions that can be targeted. These transformations include especially the loop splitting and unrolling, which enable to allocate different precisions for different iterations, and thus to improve the tuning process. We show the advantages of this approach on a representative set of iterative programs.**

Keywords: floating-point arithmetic, mixed-precision, dynamic precision tuning tool, multiple-precision, loop splitting and unrolling, delta-debugging algorithm

I. Introduction

The representation of real numbers in computer programs is essential for scientists in many fields such as numerical analysis, physics, and engineering. For example, real numbers are used to represent a wide range of physical quantities, such as temperature, distance, and velocity. However, representing such numbers in a computer program can be challenging due to the limited precision of digital computers.

Nowadays, floating-point arithmetic is widely used to represent real numbers on computer systems. For this purpose, the IEEE-754 2019 standard provides four floating-point formats, i.e. binary16, binary32, binary64, and binary128 [1]. Other non IEEE formats are also available, like bfloat16 developed by Google, TensorFloat-32 by NVidia, or Posit proposed by Gustasfon [2]. All these offer different levels of accuracy. Given these formats, the difficulty now for the developers is to choose the appropriate format for each floating-point element of a program to satisfy an accuracy constraint. Consequently, from an accuracy point of view, they use most of the time the more accurate format directly available on the targeted hardware architectures. Yet, from a performance point of view, the length of the type chosen for floating-point computations can have a great impact on the execution time of the program. For example, loading binary32 data from memory is cheaper than loading binary64 data, and computing with binary32 data is cheaper, as well. This is particularly true on SIMD units since it is possible to pack twice as many binary32 than binary64 data for the same register size.

Several works have already shown that sometimes lowering the precision level of some variables, functions, or instructions in numerical programs does not cause too much loss of final accuracy, but it allows the program to gain speed. For example, Baboulin *et al.* showed in [3] that such mixed-precision programs can achieve similar accuracy to the original binary64 precision program while being significantly faster and reducing memory pressure. However, this work was done by hand, and guided by the complexity of the subroutines in use. And, in the general case, choosing the appropriate format for each data or each instruction in a program is a difficult task, which justifies the growing interest in auto-tuning tools.

In the last decade, many research projects are thus interested in auto-tuning for floating-point precision in numerical programs. In this context, two approaches coexist. On the first side, static approaches: Such tools analyze the program without executing it to determine a precision requirement of each variable in order to satisfy an accuracy constraint. Let us cite for example Daisy [4], [5]. This tool combines affine and interval arithmetics together with rewriting optimizations and SMT-solving to produce finite precision programs that fulfill accuracy requirements. In addition, it outputs correctness certificates that can be checked by Coq or HOL4. FPTuner [6] produces mixed-precision programs too, and it determines rigorous errors bounds by using Symbolic Taylor Expansions [7]. Another alternative is proposed by Martel in [8]. This tool relies on a 2-step static analysis (forward and backward) done by abstract interpretation to determine the precision of each variable. It outputs mixed-precision programs, as well, and constraints that can be afterward checked by an SMT solver. These techniques have also been used in the SMT-based version of POP [6]. These approaches give good results but remain limited for large programs and complex structures like loops. On the other side, dynamic approaches: Most of them rely on a trial-and-error strategy, where a step is to compare the results of both the original program and the transformed one to estimate the accuracy loss. This is for example implemented in Craft [9], [10] or Precimonious [11]. The former uses a breadth-first search strategy to explore various mixed-precision configurations, while the latter relies on the delta-debugging

```
int main(void) {
  double S = 0.;
#pragma clang loop split_optimization(enable)
  for (int i = 1; i <= 1000; i++)
    S = S + 0.01;
  printf("S = %.20e", S);
  // ... |S - Soptim|/|S| < 1e-6 ?
  check_reverse_rel_error(S, 1e-6);
  return 0;
}
```

Listing 1: C program computing the sum S in (1).

algorithm [12]. This heuristic algorithm works like a binary search to determine a locally maximum set of changes that allows to fulfill the accuracy requirement and to produce a mixed-precision program faster than the original one. One of the limitations of these approaches is the combinatorics linked to the number of possible transformations. To tackle this issue, two extensions of Precimonious have been proposed. Blame [13] reduces its search space by quickly exhibiting variables whose precision does not impact the final result, and HiFPTuner [14] groups dependent data requiring the same level of precision in communities and it applies transformations directly to these communities. Other alternatives also exist: for example, Promise [15] and Verrou [16] combine both Discrete Stochastic Arithmetic through the CADNA library [17] and delta-debugging to produce mixed-precision programs, while ADAPT [18] relies on automatic differentiation to determine precision requirements of inputs and intermediate variables. Recently, Pherbie [19] has been developed on top of Herbie [20], which uses rewriting rules and multiple-precision together with local error analyses to adapt the precision of sub-expressions and to produce a set of mixed-precision candidate implementations optimized for accuracy and speed. These tools allow to tackle larger problem than static ones, but they do not consider the iterative nature of the program.

Indeed less attention has generally been paid to the tuning of iterative programs. This is one of the limitations of Precimonious mentioned by its authors, and only a few attempts have been proposed in [21], [22], [23]. Hence, in this article, we focus on dynamic auto-tuning tools for iterative routines. The originality of our approach is to use static loop transformations to increase the number of instructions appearing in the program, improving consequently the tuning process by allowing the allocation of different precisions at different iterations.

The main contributions of this article are the following:

- A dynamic auto-tuning tool targeting iterative routines, and that relies on three modules: static loop transformations to increase the combinatorics of possible precision changes, the delta-debugging to find the most relevant ones, and the `fp2mp` module that duplicates floating-point computations with their equivalent in multiple-precision in the original program in order to evaluate the impact on numerical accuracy of tested configurations,
- And a set of examples, to show how our tool allows to improve the result of the tuning process to the detriment of an increase in the tuning time.

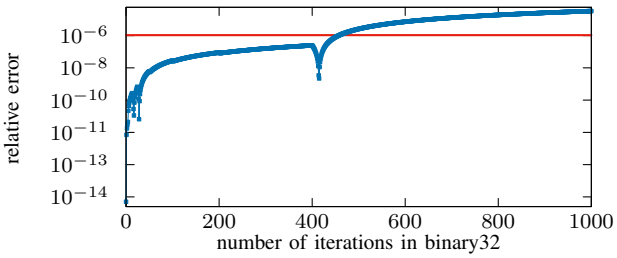


Figure 1: Relative error for the computation of S in (1), according to the number of binary32 iterations.

The article is organized as follows: It starts by a motivating example in Section II to show the interest of our approach. Then Section III presents our tool, and in particular the different modules it relies on. Section IV gives some experimental results before a conclusion in Section V.

II. Motivating example

To demonstrate the usefulness of our approach, let us consider the program in Listing 1,[1] that computes the sum

$$S = \sum_{i=1}^{1000} 0.01 = 10. \tag{1}$$

and where all the computations are performed in binary64 arithmetic. It returns `S = 9.9999999999983124610`.[2] If the program was implemented using binary32 arithmetic, the result would be `S = 10.000133514404296875`00,[3] meaning with a relative error of about 10^{-5} compared to the result in binary64.

Now, when dealing with auto-tuning tools, the question is the following: What data or what instructions can be used in lower precision (i.e. binary32) to achieve a result as accurate as the one obtained using only the higher precision (i.e. binary64) for a given threshold? For example, let us consider relative error and a threshold of 10^{-6}. The program contains two data to be tuned (the variable `S` and the constant 0.01), and existing approaches will give no solution because they will detect only three possible transformations, which all give a result with a relative error higher than 10^{-6} compared to binary64 result. In our case, we aim at adapting the precision of instructions: it is worth it since only one possible transformation is detected resulting in the binary32 result.

However, some of these 1000 iterations, whose body contains only one instruction each, could be performed in binary32 arithmetic, while keeping an acceptable output accuracy. For example, let us see what happens if the program would perform the first iterations in binary32, and the last ones in binary64, only. Figure 1 shows the relative error of such a result compared to the full binary64 result, according to the number of binary32 iterations. We can observe that at most 458 iterations could be performed in binary32 before the relative error exceeds the threshold of 10^{-6}.

[1] The program is here decorated to be used by our tool.
[2] Exact value in binary64 is $5629499534213025 \cdot 2^{-49}$.
[3] Exact value in binary32 is $20971800 \cdot 2^{-21}$.

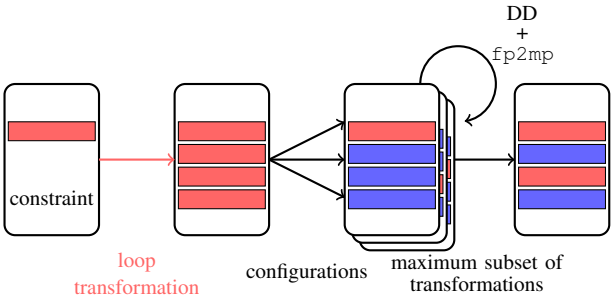


Figure 2: Workflow of our auto-tuning tool.

Now to push this idea to the limit, we can fully unroll this loop, thus resulting in a program of 1000 distinct instructions. Then by using the delta-debugging algorithm presented in Section III-D, we can isolate 526 instructions out of these 1000 that can be transformed while maintaining the relative error below the threshold of 10^{-6}.

Both of these results offer a huge improvement, but they remain untrackable by existing tools. Therefore this justifies the interest of auto-tuning tools that consider the iterative nature of programs, like the one presented in this article.

III. Auto-tuning tool for iterative programs

Our goal is to develop a dynamic auto-tuning tool that considers the iterative nature of scientific computing routines. The originality of our approach is that it relies on compilation techniques, including loop splitting and unrolling, to increase the number of instructions to be targeted in the program, without modifying its semantics, and consequently to increase the number of possible configurations. Then these configurations are exposed to the delta-debugging algorithm, to isolate a more relevant one, meaning one that allows to keep an acceptable output accuracy. And each time a configuration is considered, its impact on output accuracy is evaluated using multiple-precision through the `fp2mp` tool.

This section starts with an overview of its general workflow. Then it explains the different components of this flow, before going back to motivating example.

A. General workflow

Our tool is a command-line tool written in Python, and that requires a program together with a constraint. In this program, the loop to be tuned is pointed out by a new pragma we have introduced in the LLVM front-end, as shown in Listing 1. Moreover, the command-line gives also to the user the capability of selecting the formats including standard floating-points formats (i.e. binary16, binary32, and binary64) and bfloat16, as well as the strategy (i.e. splitting or unrolling) used in the tuning process and detailed in Section III-B. Then it works iteratively, by reducing the format of each instruction to the format immediately below it in the list of selected formats. It then repeats this step, as long as format adaptations are possible. Thus, it allows to mix several formats, and not only two, in the same program.

The general flow of our tool is given in Figure 2. As most dynamic tool, ours uses a trial-and-error strategy. Precisely, it proceeds in 3 main steps:

1) First, it starts by applying loop transformations on the input. These transformations are guided by hints given by the user through command-line parameters, and they currently include loop splitting and unrolling. This step outputs a modified LLVM intermediate representation.
2) Second, for each floating-point instruction of the programs, it inserts in this LLVM IR the equivalent computation in multiple-precision. This is done using the `fp2mp` tool, as explained in Section III-E. This tool outputs also a list of MPFR objects, whose precision can be further modified. This is used to build the configuration sets as explained in Section III-C.
3) Third, using the delta-debugging algorithm, we determine a maximal subset of transformations that allows to still obtain an acceptable output. This is explained in Section III-D.

At the end of this process, we do not generate a new optimized program, but we give some hints to the user on how to produce it through a list of acceptable transformations.

B. Static loop transformations

The originality of our approach comes from the static loop transformations applied on the original program. Their objective is to increase the number of possible transformations. For doing this, our tool leverages the LLVM capabilities of transforming programs.

LLVM is a set of tools and libraries that are used for compiler construction. One of its key features is its intermediate representation, which serves as a high-level assembly language that is portable across different platforms and architectures. The LLVM IR is designed to be easily optimized by various transformations during multiple passes. These passes allow developers to perform various optimizations on the program, such as unrolling loops, inlining functions, and vectorizing programs. These optimizations can help to improve the performance of a program, particularly for programs that have a high degree of iterative or repetitive behavior.

In this project, we use the capability of unrolling loops available in LLVM, and the one of splitting loops developed by Revy [24]. These are both loop transformations that make instructions appeared in the program, but without modifying its semantics. Here we called *strategy* a combination of a loop transformation and a factor. For example, on the program of Listing 1, the strategy of unrolling loop by a factor of 2 gives the following program,

```
for (int i = 1; i <= 500; i++) {
  S = S + 0.01;
  S = S + 0.01;
}
```

while the one of splitting the loop by a factor of 2 gives the following one.

```
for (int i = 1; i <= 500; i++)
  S = S + 0.01;
for (int i = 501; i <= 1000; i++)
  S = S + 0.01;
```

This strategy is given to LLVM core by inserting appropriate metadata in the LLVM IR.

These transformations allow for determining two different patterns of transformations. Indeed the unrolling allows the detection of some transformations appearing regularly throughout the iterations. On the other hand, the splitting allows detecting transformations that are relevant on a subset of iterations, only (e.g. at the beginning, at the end, ...).

Note that this approach is a bit antagonist to the existing ones. Indeed the current trend is to reduce the combinatorics of targeted instructions to speed up the process: the fewer the possible instructions, the less configurations to test, and the faster the tuning process. This is actually done in Blame or HiFPTuner that reduce the combinatorics compared to Precimonious, for example. Conversely, in our approach, we increase this combinatorics, but in a reasonable way. Doing this way, we certainly increase the time necessary for the process to converge, but we significantly improve the quality of this process, as illustrated in Section IV.

C. Building transformation sets

This section deals with the way used to build the set of possible transformations. This set is given to the delta-debugging to determine the final configuration.

Once the loop transformation is performed, and the `fp2mp` is run on the modified LLVM IR, we get a list of MPFR objects for each function. This list corresponds to all the floating-point elements of the input program (i.e. variables, constants, and operations). It is given in a JSON file, whose entries are like the one below.

```
{
  "name": "add.mpfr",
  "fp_name": "add",
  "fp_type": "double",
  "precision": "53",
  "tuned": "true",
  "location": {"line": "12", "column": "11"},
  "op_code": "fadd",
  "block": "for.body",
  "loops": [ {"name": "for.cond", "depth": "1"} ]
}
```

Hence for each of them, it contains especially its floating-point type and its precision, its location, and possibly the loop it belongs to in the program.

Using this information, it is possible to build a list of possible transformations. In our case, we target floating-point instructions. Hence a transformation is a tuple composed of an instruction and a list of formats, all lower than the format of the instruction itself and ordered by decreasing precision. They correspond to the formats to which the tool will try to reduce the instructions. For example, if the user ask for using binary16, binary32, and binary64, for the addition `add` below, we will have the following list of possible transformations.

```
add -> [b32, b16]
```

This means that the tool will start by trying to reduce the format of the `add` instruction to the binary32 format. If it succeeds, it will then try to reduce it to the binary16 format.

For performance purposes, when the splitting strategy is used, it can make sense to use the same format for all the instructions in a loop. In this sense, we implemented another way to define transformation sets, by grouping all the instructions of a loop body in the same set and by assigning a unique set of possible target formats to this group. This is used for example in Sections IV-B and IV-C.

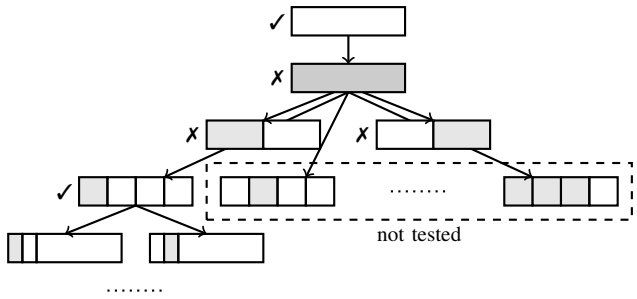


Figure 3: Delta-Debugging workflow.

D. The delta-debugging algorithm

The delta-debugging algorithm, developed by Zeller and Hildebrandt, was initially designed for the simplification of test cases. This is a process of reducing the complexity of a test case while still maintaining the ability to expose a bug. This is useful in debugging because it can help to isolate the cause of a bug, making it easier to fix. Its workflow is given in Figure 3. It is implemented in a tool called *ddmin* which is widely used and it has been proven to be effective in reducing the size of test cases for many types of bugs. It works like a binary search to determine a 1-minimal set $\Delta_\checkmark$ of changes that reproduces the bug. This means that removing only one change from $\Delta_\checkmark$ makes the bug go away.

Several works have been published to adapt it to specific use cases. In the context of auto-tuning, it has been adapted to be used by other tools, like for example Precimonious or Promise. In our case, we use *ddmax*, a variant of the delta-debugging *ddmin* routine [25]. Given a set Δ of possible transformations, its goal is to maximize the subset $\Delta_\checkmark$ of the input set Δ that makes the initial program still computing an acceptable output. At the end of the algorithm, $\Delta_\checkmark$ is 1-maximal, which means that adding one transformation in $\Delta_\checkmark$ will make the accuracy test failed. Actually *ddmax* uses the same techniques as *ddmin*: maximizing $\Delta_\checkmark$ can be done by systematically minimizing $\Delta - \Delta_\checkmark$, starting with large differences and then smaller and smaller ones, until every remaining differences would cause $\Delta_\checkmark$ to produce unacceptable output.

From an implementation point of view, our tool relies on the DD Python module.[4] This provides a class that implements the delta-debugging algorithm. In order to adapt it to our needs, we have overloaded the `test()` method. It is in charge of testing whether a configuration of transformations allows to satisfy the input constraint or not. For doing this, it modifies the precision of the corresponding MPFR objects in the LLVM IR using the `update` module of `fp2mp`.

E. The `fp2mp` verification tool

Dynamic auto-tuning tools often rely on a trial-and-error strategy, for a dataset for which a reference result is known.

[4]See https://github.com/grimm-co/delta-debugging/.

Particularly, they iteratively apply transformations on the input until the accuracy test fails. For doing this, this test procedure uses a module to evaluate the impact on the output accuracy of some transformations on the input program.

In our case, we use multiple-precision computations. Particularly, our auto-tuning process relies on the `fp2mp` tool. This tool is designed for experimentation with floating-point precision in computer programs. It works by duplicating in the LLVM IR each floating-point instruction used in a program with its equivalent in multiple-precision through the MPFR library. This allows the user to define the precision they desire, and to evaluate the impact of these precision modifications on the behavior of the program.

The interest of `fp2mp` in our approach is that this module offers the capability of adapting the precision of certain computations only. Even if it works on LLVM IR, for a sake a clarity, let us see how it would work at a C level. For example, after having unrolled the loop by a factor of 2, Listing 1 would be transformed as follows where `Smpfr` holds the equivalent value of `S` but in MPFR.

```
    double S = 0;
    mpfr_t Smpfr;
    mpfr_init2(Smpfr, 53);
    mpfr_set_d(Smpfr, 0., MPFR_RNDN);
    for(int i = 0; i <= 1000; i++) {
      S = S + 0.01;
      mpfr_add_d(Smpfr, Smpfr, 0.01, MPFR_RNDN);
      S = S + 0.01;
      mpfr_add_d(Smpfr, Smpfr, 0.01, MPFR_RNDN);
    }
    mpfr_clears(Smpfr, (mpfr_ptr) 0);
```

Hence adapting the precision of only the second addition to the binary32 format, for example, would result in the following piece of C code, where `C1`, `C2` and `C3` are three temporary variables representing values in binary32, that is, in precision 24, to ensure the computation is done in precision 24, as well.

```
    double S = 0;
    mpfr_t Smpfr, C1, C2, C3;
    mpfr_init2(Smpfr, 53);
    mpfr_inits2(24, C1, C2, C3, (mpfr_ptr) 0);
    mpfr_set_d(Smpfr, 0., MPFR_RNDN);
    for(int i = 0; i <= 1000; i++) {
      S = S + 0.01;
      mpfr_add_d(Smpfr, Smpfr, 0.01, MPFR_RNDN);
      S = S + 0.01;
      mpfr_set(C1, Smpfr, MPFR_RNDN);
      mpfr_set_flt(C2, 0.01f, MPFR_RNDN);
      mpfr_add(C3, C1, C2, MPFR_RNDN);
      mpfr_set(Smpfr, C3, MPFR_RNDN);
    }
    mpfr_clears(Smpfr, C1, C2, C3, (mpfr_ptr) 0);
```

F. Back to the motivating example

To reinforce the interest of our tool, let us now go back to the motivating example of Section II. Using a strategy of splitting the loop by a factor of 20 would result in the following piece of program.

```
  int main(void) {
    double S = 0;
    for (int i = 1; i< = 50; i++)
      S = S + 0.01;
    for (int i = 51; i <= 100; i++)
      S = S + 0.01;
    // ...
    for (int i = 951; i <= 1000; i++)
      S = S + 0.01;
    // ... |S - Soptim|/|S| < 1e-6 ?
    check_reverse_rel_error(S, 1e-6);
    return 0;
  }
```

Hence we no longer have a single instruction, but 20 separate ones. Assuming two formats (i.e. binary64 and binary32), the combinatorics amounts to $2^{20} = 1048576$, meaning 1048576 possible configurations.

By running our tool, in this case, we found in about 10 sec. 9 instructions out of these 20 ones that could be transformed so that the result of this transformed program remains at the required threshold of 10^{-6}. This means 45% of the instructions are performed in binary32 arithmetic, which is actually a significative improvement compared to existing methods. And this result is not far from the one obtained by fully unrolling the program in Section II. However, if the unroll strategy by a factor of 20 would have been used, only 1 instruction out of the 20 would have been isolated to be relevant, that is, only 5% of the instructions.

Therefore, the quality of the result directly depends on the strategy used. However, finding the best strategy for a given problem is a difficult task that we do not discuss in this article, we leave this choice to the user.

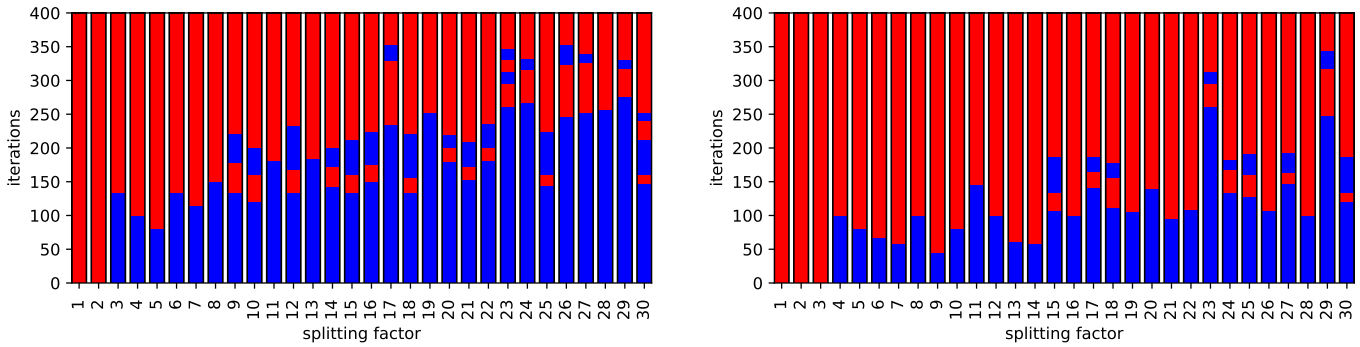
IV. Experimental results

In this section, we presents some experimental results to show the interest of our analysis techniques.

A. Impact of different strategies

In this first experiment, we use 5 iterative programs: the *sum* problem from Listing 1 (1000 iterations), the *riemann* and *simpson* methods that compute the integral of a function (400 and 1000 iterations, respectively), the *arclength* algorithm inspired from [11] and that computes the arc length of a function (1000 iterations), and finally the *nbody* problem inspired from [26] that predicts the individual motions of a group of celestial objects interacting with each other gravitationally (100 iterations). All these programs are implemented in binary64 floating-point arithmetic, and we consider reducing the precision of instructions to the binary32. For doing this, for each problem, we ran our auto-tuning tool with different strategies and different thresholds. For all the problems, the threshold considered is the relative error of the result computed by the transformed program compared to the one in binary64 (which is considered as the "exact" result), but for *nbody* where the threshold is the absolute distance between the 3D coordinates computed by the transformed program and the ones by the original one. Tables I and II show the impact of using our approach. In these tables, "delta-debugging" means the delta-debugging is applied without loop transformations, like other tools would have done. Moreover, for each problem, these give the number of binary64 and binary32 instructions, together with the transformation percentage.

We observe that in all cases, applying static loop transformations allow to improve the quality of the tuning result. For example, let us consider the *arclength* problem and the *splitting* strategy. For a threshold of 10^{-8}, the delta-debugging alone found 5 out of the 11 instructions of the main loop



(a) threshold of 10^{-8}

(b) threshold of 10^{-9}

Figure 4: Iteration patterns for the *riemann* method and different splitting factors.

that can be transformed, i.e. 45.5%. Splitting the loop into 50 subloops increases the combinatorics: now 550 instructions are available in the program, i.e. 2^{50} possible configurations. Applying the delta-debugging on it allows to achieve 87.1% of transformations, i.e. 479 instructions among these 550 ones, which is a huge improvement. And the improvement is even better for sharper thresholds. Still on *arclength* problem and *splitting* strategy by a factor of 50, for a threshold of 10^{-11}, no solution is found by applying only the delta-debugging, while our solution found 59 instructions out of the 550 to be tuned, i.e. 10.7%. The same holds for other problems, and the improvement can be even better, like for *riemann* method and a threshold of 10^{-8}, where the *unrolling* strategy is used with a factor of 50. In this case, almost all the instructions are transformed (i.e. 99.2%), while the delta-debugging alone isolates only 40% of the instructions.

However, whatever the strategy, the higher the factor, the greater the number of instructions in the transformed program, and therefore the more important the combinatorics. In this case, the tuning time also increases. For example, still on the example of *arclength* problem and *splitting* strategy, the tuning time goes from a few seconds to several tens of minutes for a factor of 50 (i.e. 19 min. for a threshold of 10^{-8} to 48 min. for 10^{-11}). But this can be contained: for example, for a factor of 20, the tuning time does not exceed 10 min.

B. Same transformations for the whole loop body

From a performance point of view, it is better to have instructions of the same format in a given loop body. For example, it helps in activating some compiler optimizations like vectorization [14]. Hence, in this experiment, we ran our auto-tuning tool by forcing all the instructions of a same loop to be in the same format. (This does not make sense for *unrolling* strategy, but only for *splitting* strategy.) We did this experiment for the *riemann* problem, for a splitting factor ranging from 1 to 30. Figure 4 shows the results for a threshold of 10^{-8} and 10^{-9}, where the iterations in binary32 are in blue, and the ones in binary64 are in red.

As expected, we observe that the value of the splitting factor impacts the number of instructions transformed. For example, in Figure 4a, for a threshold of 10^{-8}, and for a factor in $\{1, 2\}$, no transformation is found and all the iterations are in binary64 arithmetic. For a factor of 3, the first third of iterations (i.e. 133 iterations) are transformed in binary32, while the two last thirds remain in binary64. And this ratio increases with the value of the splitting factor. The same holds for a threshold of 10^{-9} in Figure 4b, but since the threshold is sharper, the number of binary32 iterations is lower.

Note in this experiment that, as mentioned in Section III-B, the *splitting* strategy helps to find patterns where the transformations appear at the beginning of all the iterations.

C. Auto-tuning of unbounded loops

In the previous experiments, the constraint was on the accuracy of the output. The constraint can also relate to the number of iterations, for example, which is particularly interesting in the case of unbounded loops. In this third experiment, we consider the *conjugate gradient* given in Algorithm 1. This is a well-known algorithm to solve the linear system $A \cdot x = b$ in the case of symmetric definite-positive matrices. It iterates as long as the residual $\|r_k\|$ is greater than a given threshold. And it is actually known to be extremely sensitive to the computation precision, which impacts directly this number of iterations.

For this last experiment, we use the `494_bus` matrix from the Suite Sparse Matrix Collection [27], where b is so that the solution x is a vector of ones, and we ran the algorithm with a threshold of 10^{-6} on the residual. Implemented in binary64, it requires 1315 iterations to converge, while in binary32 this number of iterations scales to 2494. The objective was to find in the binary64 version of the program some iterations that

Algorithm 1 Conjugate Gradient method.

$r_0 := p_0 := b - Ax_0$, and $k = 0$
while $\|r_k\| \geq \epsilon$ and $k <$ maxiter **do**
  $\alpha_k := \frac{r_k^T r_k}{p_k^T A p_k}$
  $x_{k+1} := x_k + \alpha_k p_k$
  $r_{k+1} := r_k - \alpha_k A p_k$
  $\beta_k := \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
  $p_{k+1} := r_{k+1} + \beta_k p_k$
  $k = k + 1$
end while

Program	Threshold	delta-debugging				split / factor = 10				split / factor = 20				split / factor = 50			
		b64	b32	%	h:m:s	b64	b32	%	h:m:s	b64	b32	%	h:m:s	b64	b32	%	h:m:s
sum	1e-5	1	0	0.0	0:00:01	5	5	50.0	0:00:06	9	11	55.0	0:00:14	22	28	56.0	0:00:43
	1e-6	1	0	0.0	0:00:01	6	4	40.0	0:00:07	11	9	45.0	0:00:12	27	23	46.0	0:00:48
	1e-7	1	0	0.0	0:00:01	9	1	10.0	0:00:09	17	3	15.0	0:00:17	43	7	14.0	0:00:59
	1e-8	1	0	0.0	0:00:01	10	0	0.0	0:00:09	20	0	0.0	0:00:17	48	2	4.0	0:01:08
riemann	1e-8	3	2	40.0	0:00:04	15	35	70.0	0:00:41	16	84	84.0	0:00:59	10	240	96.0	0:01:44
	1e-9	3	2	40.0	0:00:04	22	28	56.0	0:00:53	42	58	58.0	0:01:52	14	236	94.4	0:02:04
	1e-10	4	1	20.0	0:00:04	29	21	42.0	0:01:14	63	37	37.0	0:02:16	19	231	92.4	0:02:22
	1e-11	5	0	0.0	0:00:04	44	6	12.0	0:01:05	80	20	20.0	0:03:17	19	231	92.4	0:02:31
arclength	1e-8	6	5	45.5	0:00:10	31	79	71.8	0:02:32	40	180	81.8	0:04:54	71	479	87.1	0:19:05
	1e-9	8	3	27.3	0:00:10	64	46	41.8	0:02:48	90	130	59.1	0:07:23	131	419	76.2	0:24:57
	1e-10	9	2	18.2	0:00:10	86	24	21.8	0:03:15	135	85	38.6	0:07:00	130	420	76.4	0:31:00
	1e-11	11	0	0.0	0:00:09	105	5	4.5	0:02:28	203	17	7.7	0:08:47	491	59	10.7	0:48:38
simpson	1e-7	4	6	60.0	0:00:06	18	82	82.0	0:01:42	4	196	98.0	0:00:33	14	486	97.2	0:02:13
	1e-8	5	5	50.0	0:00:05	34	66	66.0	0:02:11	44	156	78.0	0:05:16	77	423	84.6	0:18:20
	1e-9	6	4	40.0	0:00:09	58	42	42.0	0:02:28	48	152	76.0	0:05:19	182	318	63.6	0:27:20
	1e-10	9	1	10.0	0:00:09	60	40	40.0	0:03:14	117	83	41.5	0:07:28	291	209	41.8	0:33:14
	1e-11	10	0	0.0	0:00:09	91	9	9.0	0:03:21	178	22	11.0	0:08:28	439	61	12.2	0:41:46
nbody	1e-7	9	15	62.5	0:00:19	7	233	97.1	0:03:05	4	476	99.2	0:03:22	2	1198	99.8	0:02:16
	1e-8	16	8	33.3	0:00:27	59	181	75.4	0:06:06	84	396	82.5	0:17:04	35	1165	97.1	0:34:06
	1e-9	20	4	16.7	0:00:29	130	110	45.8	0:10:32	152	328	68.3	0:28:42	176	1024	85.3	2:06:18
	1e-10	21	3	12.5	0:00:27	212	28	11.7	0:08:17	386	94	19.6	0:34:07	742	458	38.2	5:12:40
	1e-11	24	0	0.0	0:00:24	228	12	5.0	0:07:26	443	37	7.7	0:26:53	1062	138	11.5	2:34:49
	1e-12	24	0	0.0	0:00:25	237	3	1.2	0:08:09	474	6	1.2	0:21:14	1160	40	3.3	3:03:27

Table I: Auto-tuning results for *splitting* strategy.

Program	Threshold	delta-debugging				unroll / factor = 10				unroll / factor = 20				unroll / factor = 50			
		b64	b32	%	h:m:s	b64	b32	%	h:m:s	b64	b32	%	h:m:s	b64	b32	%	h:m:s
sum	1e-5	1	0	0.0	0:00:01	6	4	40.0	0:00:08	11	9	45.0	0:00:16	25	25	50.0	0:00:40
	1e-6	1	0	0.0	0:00:01	10	0	0.0	0:00:08	19	1	5.0	0:00:16	45	5	10.0	0:00:54
	1e-7	1	0	0.0	0:00:01	10	0	0.0	0:00:09	20	0	0.0	0:00:15	48	2	4.0	0:00:56
	1e-8	1	0	0.0	0:00:01	10	0	0.0	0:00:08	20	0	0.0	0:00:16	48	2	4.0	0:00:54
riemann	1e-8	3	2	40.0	0:00:04	19	31	62.0	0:00:29	11	89	89.0	0:01:09	2	248	99.2	0:02:00
	1e-9	3	2	40.0	0:00:04	25	25	50.0	0:00:46	53	47	47.0	0:02:04	43	207	82.8	0:04:39
	1e-10	4	1	20.0	0:00:04	33	17	34.0	0:00:39	88	12	12.0	0:02:17	82	168	67.2	0:10:12
	1e-11	5	0	0.0	0:00:04	44	6	12.0	0:01:02	88	12	12.0	0:02:18	213	37	14.8	0:07:36
arclength	1e-8	6	5	45.5	0:00:10	32	78	70.9	0:01:40	40	180	81.8	0:04:18	67	483	87.8	0:12:50
	1e-9	8	3	27.3	0:00:10	31	79	71.8	0:02:39	125	95	43.2	0:07:12	80	470	85.5	0:22:54
	1e-10	9	2	18.2	0:00:10	93	17	15.5	0:02:48	180	40	18.2	0:06:00	378	172	31.3	0:51:03
	1e-11	11	0	0.0	0:00:09	110	0	0.0	0:01:57	210	10	4.5	0:06:38	509	41	7.5	0:27:58
simpson	1e-7	4	6	60.0	0:00:06	10	90	90.0	0:00:39	15	185	92.5	0:01:11	23	477	95.4	0:03:09
	1e-8	5	5	50.0	0:00:05	47	53	53.0	0:01:26	28	172	86.0	0:03:25	47	453	90.6	0:04:47
	1e-9	6	4	40.0	0:00:09	56	44	44.0	0:01:24	112	88	44.0	0:03:36	179	321	64.2	0:18:18
	1e-10	9	1	10.0	0:00:09	62	38	38.0	0:01:53	117	83	41.5	0:07:01	275	225	45.0	0:18:30
	1e-11	10	0	0.0	0:00:09	95	5	5.0	0:02:23	182	18	9.0	0:06:02	413	87	17.4	0:39:05
nbody	1e-7	9	15	62.5	0:00:19	36	204	85.0	0:02:33	14	466	97.1	0:02:33	4	1196	99.7	0:07:55
	1e-8	16	8	33.3	0:00:27	66	174	72.5	0:06:19	80	400	83.3	0:10:44	58	1142	95.2	0:52:36
	1e-9	20	4	16.7	0:00:29	158	82	34.2	0:09:49	194	286	59.6	0:47:44	181	1019	84.9	3:13:32
	1e-10	21	3	12.5	0:00:27	206	34	14.2	0:09:28	395	85	17.7	0:32:58	900	300	25.0	2:33:01
	1e-11	24	0	0.0	0:00:24	228	12	5.0	0:08:29	455	25	5.2	0:22:35	1046	154	12.8	3:51:45
	1e-12	24	0	0.0	0:00:25	238	2	0.8	0:06:16	471	9	1.9	0:25:52	1176	24	2.0	1:35:16

Table II: Auto-tuning results for *unrolling* strategy.

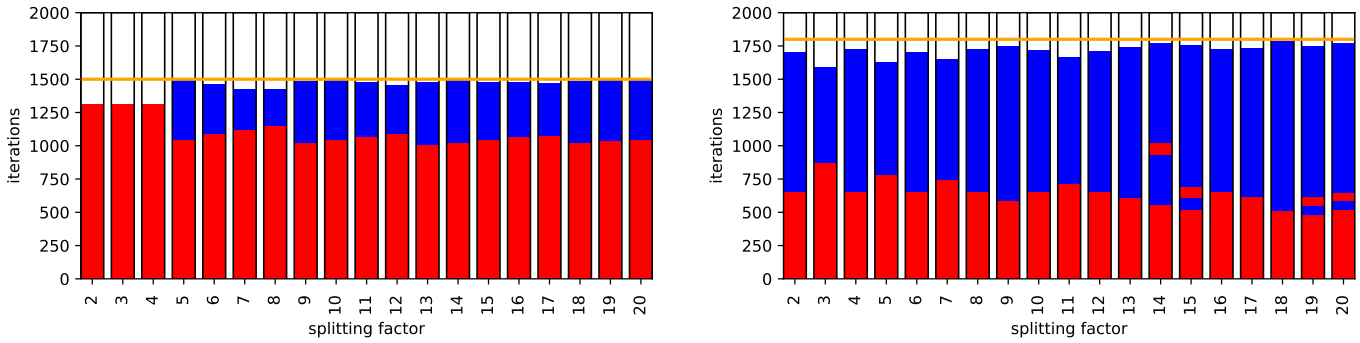
could be transformed into binary32 so that the total number of iterations remains no larger than a user-defined bound.

For doing this, we ran our tool by forcing all the instructions of the main loop to be in the same format, as in Section IV-B. And we repeated this for a splitting factor ranging from 2 to 20. Figure 5 shows the results for a number of iterations bounded by 1500 and 1800.

Our tool allows to find automatically some iterations to be transformed in binary32 arithmetic. For example, for a splitting factor of 2, we obtained two loops in binary64 arithmetic, of 657 and 658 iterations, respectively. Then if we bound the total number of iterations to 1500 (Figure 5a), no iteration can be lowered in binary32, and all the 1315 iterations remain in binary64 (in red in the figures). However, if we bound the number of iterations to 1800 (Figure 5b), the tool found that the second loop can be performed in binary32 (in blue in the figures). However, since the computation precision impacts the number of iterations, lowering the precision of instructions in the second loop makes its number of iterations increased. It scales to 1046, for a total of 1703 iterations, which is still below the bound.

V. Conclusion

This article addresses the tuning of precision in iterative routines. We have designed an auto-tuning tool that relies on compiler capabilities and not on the user action to improve the tuning process. Particularly, it applies static loop transformations on the original program to increase the number of instructions appearing in the program and thus to yield better solutions. We showed on some examples, that it helps to improve significantly the existing solutions, even in the case of unbounded loops, to the detriment of an increase in the tuning time. A direct extension would be to give the capability to user of combining both these loop transformations.



(a) number of iterations bounded by 1500

(b) number of iterations bounded by 1800

Figure 5: Increase of iteration numbers for the *conjugate gradient* and different splitting factors.

Now the research direction is threefold: First, it would be of interest to measure the runtime of result programs to evaluate the speedup delivered by our tool. Second, other static loop transformations exist, and it could be interesting to investigate their interest. For example, the loop peeling and versioning lead also to a duplication of the instructions of the loop, and they could be useful for our auto-tuning approach. Third, we have applied our techniques on small routines embracing simple loops or, in the case of the *conjugate gradient*, on the main loop. It could be interesting to study how this approach scales with the size of the loop body since instructions are duplicated, and how it behaves on nested loops.

Acknowledgment

This work was supported by the PADOC ANR project under grant n°ANR-18-CE25-0004.

References

[1] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*, 2019.

[2] J. L. Gustafson and I. T. Yonemoto, "Beating Floating Point at Its Own Game: Posit Arithmetic," *Supercomputing Frontiers and Innovations*, vol. 4, no. 2, p. 71–86, 2017.

[3] 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*, vol. 180, no. 12, pp. 2526 – 2533, 2009.

[4] A. Izycheva and E. Darulova, "On Sound Relative Error Bounds for Floating-Point Arithmetic," in 17*th Conference on Formal Methods in Computer-Aided Design*, 2017, p. 15–22.

[5] E. Darulova, A. Izycheva, F. Nasir, F. Ritter, H. Becker, and R. Bastian, "Daisy - Framework for Analysis and Optimization of Numerical Programs," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Beyer and M. Huisman, Eds., 2018, pp. 270–287.

[6] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić, "Rigorous Floating-Point Mixed-Precision Tuning," in *POPL 2017*, 2017, pp. 300–315.

[7] A. Solovyev, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan, "Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions," in *FM 2015*, 2015, pp. 532–550.

[8] M. Martel, "Floating-Point Format Inference in Mixed-Precision," in *9th NASA Formal Methods Symposium.*, 2017.

[9] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. Legendre, "Automatically adapting programs for mixed-precision floating-point computation," in *ICS'13*, 2013, pp. 369–378.

[10] M. O. Lam and J. K. Hollingsworth, "Fine-grained floating-point precision analysis," vol. 32, no. 2, 2018, pp. 231–245.

[11] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: tuning assistant for floating-point precision," in *SC'13*, 2013, pp. 1–12.

[12] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-Inducing Input," *IEEE Transactions on Software Engineering*, pp. 183–200, 2002.

[13] C. Rubio-González, 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 *ICSE 2016*, 2016, pp. 1074–1085.

[14] H. Guo and C. Rubio-González, "Exploiting Community Structure for Floating-Point Precision Tuning," in *27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Association for Computing Machinery, 2018, p. 333–343.

[15] S. Graillat, F. Jézéquel, R. Picot, F. Févotte, and B. Lathuilière, "Auto-tuning for floating-point precision with Discrete Stochastic Arithmetic," *Journal of Computational Science*, vol. 36, p. 101017, 2019.

[16] F. Févotte and B. Lathuilière, "Debugging and optimization of HPC programs with the Verrou tool," in *International Workshop on Software Correctness for HPC Applications (Correctness)*, 2019, pp. 1–10.

[17] F. Jézéquel and J.-M. Chesneaux, "CADNA: a library for estimating round-off error propagation," *Computer Physics Communications*, pp. 933–955, 2008.

[18] H. Menon, M. O. Lam, D. Osei-Kuffuor, M. Schordan, S. Lloyd, K. Mohror, and J. Hittinger, "ADAPT: Algorithmic Differentiation Applied to Floating-Point Precision Tuning," in *International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 2018.

[19] B. Saiki, O. Flatt, C. Nandi, P. Panchekha, and Z. Tatlock, "Combining Precision Tuning and Rewriting," in *28th IEEE Symposium on Computer Arithmetic (ARITH 2021)*, 2021, pp. 1–8.

[20] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," in *PLDI'2015*, 2015, pp. 1–11.

[21] Y. Chatelain, E. Petit, P. De Oliveira Castro, G. Lartigue, and D. Defour, "Automatic Exploration of Reduced Floating-Point Representations in Iterative Methods," in *25th International Conference Euro-Par 2019 Parallel Processing*, 2019, pp. 481–494.

[22] H. Brunie, C. Iancu, K. Z. Ibrahim, P. Brisk, and B. Cook, "Tuning Floating-Point Precision Using Dynamic Program Information and Temporal Locality," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–14.

[23] R. Gu, P. Beata, and M. Becchi, "A Loop-Aware Autotuner for High-Precision Floating-Point Applications," in *International Symposium on Performance Analysis of Systems and Software (ISPASS 2020)*, 2020, pp. 285–295.

[24] G. Revy, "Analyzing the impact of floating-point precision adaptation in iterative programs," in *2021 IEEE 28th Symposium on Computer Arithmetic (ARITH)*, 2021, pp. 25–32.

[25] L. Kirschner, E. Soremekun, and A. Zeller, "Debugging Inputs," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 75–86.

[26] D. Ben Khalifa and M. Martel, "A study of the floating-point tuning behaviour on the n-body problem," in *Computational Science and Its Applications – ICCSA 2021*. Springer International Publishing, 2021, pp. 176–190.

[27] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software*, vol. 38, 2011.