Light-FP: Analyze Floating-Point Error in a Highly Condensed Approach

Jiazhi Mi
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
mijiazhi22s@ict.ac.cn

Li Chen[*]
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
Laboratory for Advanced Computing and Intelligence Engineering
Wuxi, Jiangsu, China
lchen@ict.ac.cn

Haoyu Wang
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
wanghaoyu21s@ict.ac.cn

Ruixiang Gao
Shandong University of Science and Technology
Qingdao, Shandong, China
gaoruixiang665@gmail.com

Hongze Zhang
Shandong University of Science and Technology
Qingdao, Shandong, China
zhanghongze1201@163.com

Ronghong Shen
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
shenronghong23s@ict.ac.cn

Kai Lin
Beijing Institute of Technology
Beijing, China
omg_link@qq.com

You Fu
Shandong University of Science and Technology
Qingdao, Shandong, China
fuyou@sdust.edu.cn

Huimin Cui
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
University of Chinese Academy of Sciences
Beijing, China
cuihm@ict.ac.cn

Abstract

Approximate computing is emerging as a promising paradigm of High-Performance Computing (HPC) to increase application performance, with mixed-precision computing standing out as a popular approach within this domain. However, mixed precision tuning is challenging due to the vast search space of mixed precision configurations. The floating-point error sensitivity profile is a technique to quantify the error contribution of each reduced precision operation to the final output error, which can dramatically reduce the search space of mixed precision tuning of applications. To control time complexity, the error sensitivity profile is currently built upon reverse-mode automatic differentiation; however, its high memory demands render this approach impractical for large HPC workloads. This paper proposes a novel error analysis algorithm with much lower space complexity, by leveraging forward-mode automatic differentiation combined with just-in-time error source aggregation. Furthermore, a new

[*]Corresponding Author

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3730432

error model is introduced to capture dependencies among assignments within the same error source group, addressing a key limitation of existing techniques that assume floating-point assignments to be independent of each other, which is an assumption sufficient only for smaller-scale cases. We implement this approach within the LLVM compiler infrastructure, as an easy-to-use source-code transformation tool, Light-FP. Evaluations conducted on applications from the HPC-MixPBench benchmark suite demonstrate Light-FP's superiority over the state-of-the-art tool in terms of accuracy, runtime, and memory efficiency.

CCS Concepts

• **Mathematics of computing → Arbitrary-precision arithmetic**; • **Software and its engineering → Dynamic analysis**.

Keywords

Floating-Point Arithmetic, Round-off Error, Precision Tuning, Dynamic Program Analysis

ACM Reference Format:
Jiazhi Mi, Li Chen, Haoyu Wang, Ruixiang Gao, Hongze Zhang, Ronghong Shen, Kai Lin, You Fu, and Huimin Cui. 2025. Light-FP: Analyze Floating-Point Error in a Highly Condensed Approach. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3721145.3730432

1 Introduction

In the post-Moore era, approximate computing is gaining prominence as an effective strategy to enhance application performance. This approach leverages the fact that many applications can tolerate a certain level of error, allowing for a trade-off between accuracy and performance. Within this field, reduced floating-point precision, or mixed-precision computing, has recently gained popularity. Many studies [2], [12], [1], [4] have shown that employing multiple levels of precision within a single program can significantly boost the performance of scientific applications.

Users should identify potential opportunities by pinpointing error-insensitive variables within the given application. They should then assess the precision degradation of mixed-precision configurations, evaluate the performance impacts, and implement the necessary code transformations. Among these steps, precision verification is the most challenging. Aside from costly trials, it is difficult for programmers to determine whether the errors introduced by a given configuration fall within acceptable thresholds. Domain experts may have extensive knowledge of an algorithm's numerical behavior, yet grasping the subtle details of floating-point rounding errors remains a challenge, especially for large-scale programs. Mixed precision tuning manually is even more difficult for ordinary users, and becomes increasingly infeasible for HPC programs with multiple modules.

Various automatic tools are developed to evaluate approximation errors. Static approaches using interval analysis or Taylor series approximation provide rigorous bounds for FP errors, but they do not scale very well and are limited to small programs currently [29], [8]. Dynamic analysis is more practical than static approaches. There are mainly two main categories here, search-based methods and error sensitivity profile approach. Search-based approaches [25], [16] are black-box methods, that explore various mixed-precision configurations by running the program multiple times to identify the optimal configuration that meets the error threshold. Since the size of the state space is exponential to the number of locations where the precision can be changed, different search strategies, combined with clustering algorithm are studied to reduce the search space, but the number of trials is still large [24]. The error sensitivity profile approach [21], [27] does not evaluate the accuracy of each configuration directly, but provides information on the sensitivity of the output with respect to variable approximations in one single run. Based on the sensitivity profile, the mixed precision search space can be largely reduced, and a valid, sub-optimal configuration can even be gained, without further trials. They are based on Automatic Differentiation (AD), and Taylor series approximation is introduced to model the error contribution of each error source to the final output error. ADAPT, as a representative, gains attention from many users and has been incorporated with several software tools together [18]. However, these error sensitivity tools are not accurate enough, and have high memory overhead for applications with many iterations, restricting their applicability to more HPC workloads or to platforms with limited memory.

We propose Light-FP, an efficient and highly condensed approach for analyzing approximation errors in HPC applications. Different from traditional algorithms, Light-FP is built upon forward mode automatic differentiation and error source aggregation with much lower space complexity. Besides, to overcome the drawback of existing state-of-the-art tools that assume floating-point assignments to be independent of each other, we propose a new error model to model dependencies among assignments, and prove that the improved algorithm can get accurate error estimation under the first-order Taylor series approximation. We build Light-FP using the LLVM compiler infrastructure and evaluate its effectiveness in precision tuning.

The primary contributions of this paper are as follows:

- A scalable and efficient algorithm, to analyze floating point errors due to lowering the precision of variables, based on forward mode automatic differentiation and just-in-time error accumulation. The algorithm's time and space overhead is merely a constant multiple of the original program's.

- A new error model is introduced to catch dependencies among assignments within the same error source group.
- Light-FP, a tool to realize the above algorithm. We put forward its implementation and optimizations. Adjoint index tracing substitutes hashmap retrievals with low-cost index calculations, while data parallelization significantly enhances the efficiency of the analysis.
- Tool evaluation through eight benchmarks, including six applications from HPC-MixPBench. It shows that Light-FP achieves much better accuracy and performance. Compared to the current state-of-the-art method CHEF-FP, Light-FP shows a significant advantage in memory consumption, achieving up to 2600x greater memory efficiency. It also offers runtime benefits for most applications, with a maximum speedup of 5.6x. In many cases, the accuracy of error estimation is several orders of magnitude better than the counterpart and quite close to the actual errors.

2 Background and Motivations

The execution trace of a given program segment can be denoted as $P = \langle T, IN, y\rangle$, where T is the dynamic assignment statement sequence, y is the output statement and IN is the set of upward-exposed floating point variables of T. In the error analysis scenario, both T and IN are error sources in P. For each floating point assignment, if we lower the precision of its result operand, a small, local truncation error is introduced, and this error will be scaled and propagated along def-use chains of P, and may finally reach the output assignment y. The FP variables in IN also brings truncation error. So we simply regard each input variables as a special kind of assignment, i.e. $IN \subset T$.

First-order Taylor series approximation is well suited to model floating point errors [29], [21], [27]. Let y be the output assignment of the program P, $t \in T$, and variable x be what t assigns to. Then, a function $y = f(x)$ exists that can represent the related part of the program. Assuming that x brings a floating point error h, the first-order Taylor series approximation yields:

$$f(x+h) = f(x) + \frac{h}{1!}f'(x) + \frac{h^2}{2!}f''(x) + ... \quad (1)$$

$$\approx f(x) + hf'(x) \quad (2)$$

Here $f'(x)$ represents the derivative of the function to x. So,

$$\Delta y = f(x+h) - f(x) \approx hf'(x) \quad (3)$$

It can be extended to the multi-variable function. For $y = f(x_1, ..., x_n)$, if each x_i has a floating point error h_i, then the output error is

$$\Delta y \approx \sum_{i=1}^{n} h_i f'_{x_i}(x_1, ..., x_n) \quad (4)$$

When performing error analysis to the program segment P, each FP assignment in P should be regarded as an input of function f, no matter if it is an upward-exposed variable of P or an ordinary FP assignment in T. The global derivatives $f'_{x_i}(x_1, ..., x_n)$ can be computed by Automatic Differentiation (AD) tools, and the local truncation error, h_i is computed using t, the variable value of the high precision program.

$$tcErr(t) = t_{low} - t \quad (5)$$

where t_{low} is the value taken when t is demoted to lower precision.

2.1 Current error model and its accuracy

In the scenario of mixed precision error analysis, each FP assignment in P is an error source by default.

The basic error model. Equation 4 is the error model adopted by [21]. It should be noted that, this is an independent, single-assignment instance approach, where local truncation error is obtained from the high precision program, and the accumulation here implies an assumption that the error of one assignment will not impact those of other assignments, which is not true in any real programs. This simplification happens because FP error analysis on correlated assignments has not yet been established [27]. Although this method performs well for many small programs, its accuracy deteriorates as the dependency chain lengthens, causing the error analysis to deviate increasingly from the actual values.

Let's see a simplified program of Simpsons, which is an iterative algorithm to approximate the integral of function $sin(\pi x)$ in a certain interval, shown in Figure 1. When computing the error contribution of the source-level variable $s1$, four source-code level assignments should be considered (as an input of the function, a special error source exists at the entry of the function), and the error contributions of totally of 2000002 dynamic assignments should be evaluated and accumulated (when n=1e6). Since $s1$ is a reduction variable, different assignments form a long error propagation chain and independent single-assignment error analysis significantly underestimates the output error. When n=1e6, the estimated error of $s1$ is 8.1666e-08, which is 5 orders of magnitude smaller than the actual one (2.6435e-03). Similarly, the estimated error of x is 2.3975e-07, while the actual error is 3.5069e-02.

An improved error model. To remedy the imprecise drawback of the above error model, CHEF-FP [27] modified the

```
double simpsons_kernel(double x, double s1,
    double pi, double h){
    for (int i = 0; i < n; i++) {
        x = x + h;
        s1 = s1 + 4.0 * sin(x * pi);
        x = x + h;
        s1 = s1 + 2.0 * sin(x * pi);
    }
    s1 = s1 * h * pi / 3.0;
    return s1;
}
```

Figure 1: Long inter-variable dependence chains (in Simpsons) that current error models neglect.

model to get the absolute value of each assignment's error estimation before summing up them together,

$$\Delta y \approx \sum_{i=1}^{n} \|h_i f'_{x_i}(x_1, ..., x_n)\| \tag{6}$$

to obtain a loose upper bound of the estimation.

Fortunately, in this case, formula 6 gets a better estimation. However we will show in the evaluation section that it does not guarantee a conservative result, and it incurs severe overestimates in many cases, making it difficult to find a suitable mixed-precision configuration.

Can we find a new error model that can cover data dependence among correlated assignments?

2.2 Reverse Mode AD-based Approach and its Space Complexity

Automatic Differentiation tools can create a new code that computes derivatives for a given program segment that executes a function with meaningful differentiable properties. AD tools usually support two modes (in different computing orders), forward mode and reverse mode. When the function has more inputs than outputs, the reverse mode is more efficient in computation time.

For mixed precision error analysis, each FP assignment in P can produce an error, so total $O(|T|)$ partial derivatives are needed, where $|T|$ is the total number of dynamic assignments in P, and is a huge number. In this case, each FP assignment is basically an input for the AD process. The time complexity of forward mode AD is unaffordable, and reverse mode AD seems to be the only choice. But the memory requirement of reverse mode AD is in proportion to $|T|$ in the worst case.

To control peak memory usage, recursive checkpointing along with re-computation is developed to make a trade-off between space and time, in which the basic reverse-mode AD is only scheduled on small segments of the code. And there is no optimal placement exists in general[14], users are encouraged to specify program intervals for checkpoints by hand, and some tools such as Tapenade checkpoint all subroutine calls by default. Different checkpointing schedules are put forward. Treeverse [11] targets certain loops using a binomial partitioning scheme, while [28] can checkpoint arbitrary computations in a divide-and-conquer fashion at the level of the language implementation without user annotation. Checkpointing is both complex and surely harmful to runtime performance.

Can we develop a lightweight, forward AD-based, efficient error estimation algorithm that has low complexity both on time and on memory requirements?

3 Source Aggregated Floating-Point Error Analysis

We put forward a source aggregated, floating point error analysis algorithm in this section. For ease of understanding and formalization, we break its main ideas into three parts. Firstly, we give a plain, forward AD-based error evaluation algorithm upon a data structure called **EPM (Error Propagation Matrix)**. Then an efficient algorithm is obtained by compressing the columns of the EPM. Thirdly, the truncation error term in the algorithm is further improved to model dependencies among assignments within the same compressed error source group.

3.1 EPM-based Error Analysis, the Naive Algorithm

In this section, we propose a new error analysis algorithm based on the error model introduced in section 2. This algorithm leverages a specialized data structure called the Error Propagation Matrix (EPM) and employs forward mode AD for error propagation. Although the algorithm does not bring about space or time benefits, it forms the basics of the upgraded versions in later sections.

Error source, EV and EPM. Each FP assignment whose resultant operand is a lower-precision candidate is considered as an error source since it generates truncation error. Following the define-use chain, these truncation errors are propagated through reachable expressions, ultimately arriving at the resultant operand of each such expression. So, each FP assignment acts as a medium for error transfer. To quantify this behavior, we assign each FP assignment an **Error Contribution Vector (EV)**, where each entry represents the error contributions from specific error sources in the program P. The length of such vectors is $|T|$. And we keep all of EVs of program P in an **Error Propagation Matrix (EPM)**.

Row compression. If each assignment has a unique EV, then EPM would require $|T|$ rows. However, such a large number of rows is unnecessary. EVs' define-use chains blindly follow that of the original program. Considering two consecutive

Algorithm 1 Error analysis with EPM

```
Input: $P = \langle T, IN, y \rangle$
Data structure: EPM, which is a $D \times |T|$ matrix
Output: EPM$_y$, which is the error contributions of $|T|$ error sources to $y$
    Initialization of EPM:
1:  $\forall$ i, $\boldsymbol{EPM_i} = \mathbf{0}$;
2:  $\forall$ iv $\in$ IN, $\boldsymbol{EPM_{iv,iv}}$ = tcErr(iv);
    EPM update for each assignment $\mathbf{t_k} : \mathbf{r} = \mathbf{f}(\mathbf{x_1}, \mathbf{x_2})$
3:  $\boldsymbol{EPM_r} = \frac{\partial f}{\partial x_1} * \boldsymbol{EPM_{x_1}} + \frac{\partial f}{\partial x_2} * \mathbf{EPM_{x_2}}$;
4:  $\boldsymbol{EPM_{r,t_k}} += tcErr(r)$;
```

assignments to the same variable, the second assignment overwrites the previous value, and at that time the EV of the first assignment becomes stale (can not propagate anymore). Therefore, different assignments to the same scalar variable can reuse the same EV space. It should be noted that different elements of the same array belong to different rows of EPM, in order to track data dependence properly. After such compression, the rows of EPM are reduced dramatically to D, which is the total number of scalar variables in the program.

The naive algorithm of EPM-based error analysis is given in Algorithm 1, Here EPM$_{var}$ represents the EV corresponding to variable var. and EPM$_{var,t_k}$ represents the specific entry in the EV corresponding to the assignment t_k. During EPM's initialization, most EVs are null vectors except for those corresponding to upward exposed input variables of P. Rounding errors, generating either from input variables or FP assignments, will propagate to more EVs along with the program execution. For each FP statement, the error update consists of two sub-steps. Without loss of generality, the current assignment statement t_k can be regarded as a function f of two variables. The first update is a whole-vector update, propagating each operand's EV to the resultant variable r, and the second update is to compute the local truncation error of this assignment and add it to a certain entry in the EV.

Traditionally, forward mode AD requires n sweeps when computing derivatives for a function with n inputs. With the help of EPM, the new algorithm propagates the errors and derivatives with respect to n inputs in the vector form, achieving the equivalent of n sweeps in a single analysis. When reaching the output assignment (the related variable is y), each entry of EPM$_y$ represents the product of the partial derivative and the corresponding truncation error for each error source. This corresponds to each item of the summation in Equation 4.

We use Simpsons in Figure 1, as an example to illustrate how EPM is evolved in Figure 2. In execution trace $P = \langle T, IN, y \rangle$, $IN = \{x, s1, pi, h\}$, the output statement y is on line 9, and T has more than $4e6$ assignments when $n = 1e6$. In this figure, red characters are used to denote the changes incurred by the current update step. $h^{\#i}, x^{\#i}$... represent the i^{th} assignment to the related variables. We use $\Delta h^{\#i}, \Delta x^{\#i}$... to represent the truncation errors from i^{th} assignment to the related variables. Additionally, $b1, b2..., d1, d2...$ are used to denote the error contribution propagated from the operands based on the whole-vector update in the Algorithm 1. At the stage of initialization, EPM will be set to zero firstly and then the truncation errors of the four input variables will be updated to EPM$_{iv,iv}$, where $iv \in IN$. For the statement numbered 2, two sub-steps are shown in detail, the whole-vector update overwrite the previous data in EPM$_x$, and then the truncation error is added to the x$^{\#2}$ column of the EPM$_x$. For $s1 = s1 + 4.0 * sin(x * pi)$, there are three FP operands, so the three related EPM rows are amplified and propagated to EPM$_{s1}$, and the new truncation error is added to the s1$^{\#2}$ column. The next two statements are just similar.

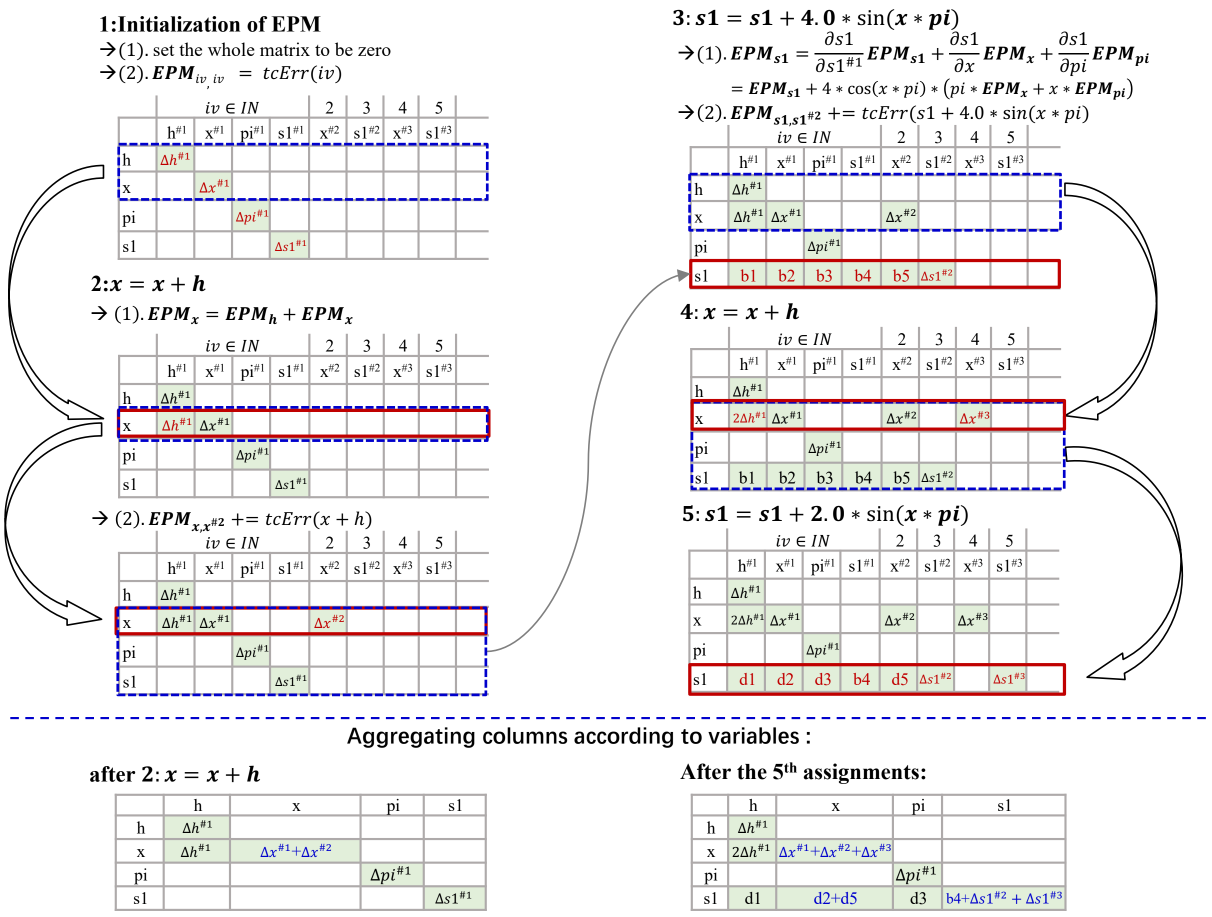
It is obvious that the above algorithm is impractical, whose time complexity is $O(|T|^2)$ and space complexity is $O(D*|T|)$. Later we will improve the algorithm by compressing the EPM columns.

3.2 Compress EPM Columns by Error Source Aggregation

In algorithm 1, EPM has $|T|$ columns where $|T|$ is huge, we want to look for a proper grouping method on EPM columns, which can keep the accuracy of error estimation.

Mixed precision tuning is commonly applied at the level of structural code blocks, variables, or arrays. Users do not want to discriminate error differences among different assignments to the same scalar variable, or the difference between each array element of the same array, since arrays are usually regarded as a whole (or several) precision regions in precision tuning. So users must accumulate error sensitivity of different assignments before precision tuning.

There are two main approaches to error accumulation, post-mortem accumulation and just-in-time accumulation. Current techniques such as ADAPT or CHEF-FP adopt the previous method, which records each dynamic assignment statement's error sensitivity during execution, and accumulates them after the analysis, according to variable names or even code hierarchy. This method provides fine-grained error information and also consumes huge memory space. The second method is based on user-provided grouping methods (also called error-source aggregation) and performs just-in-time accumulation of error sensitivity within the same group during analysis. We adopt the latter method due to its significantly lower memory requirements.



1:Initialization of EPM

→(1). set the whole matrix to be zero

→(2). $\boldsymbol{EPM}_{iv,iv} = tcErr(iv)$

	$iv \in IN$				2	3	4	5	
	$h^{\#1}$	$x^{\#1}$	$pi^{\#1}$	$s1^{\#1}$	$x^{\#2}$	$s1^{\#2}$	$x^{\#3}$	$s1^{\#3}$	
h	$\Delta h^{\#1}$								
x		$\Delta x^{\#1}$							
pi			$\Delta pi^{\#1}$						
s1				$\Delta s1^{\#1}$					

2: $\boldsymbol{x = x + h}$

→ (1). $\boldsymbol{EPM_x} = \boldsymbol{EPM_h} + \boldsymbol{EPM_x}$

	$iv \in IN$				2	3	4	5	
	$h^{\#1}$	$x^{\#1}$	$pi^{\#1}$	$s1^{\#1}$	$x^{\#2}$	$s1^{\#2}$	$x^{\#3}$	$s1^{\#3}$	
h	$\Delta h^{\#1}$								
x	$\Delta h^{\#1}$	$\Delta x^{\#1}$							
pi			$\Delta pi^{\#1}$						
s1				$\Delta s1^{\#1}$					

→ (2). $\boldsymbol{EPM}_{x,x^{\#2}} += tcErr(x + h)$

	$iv \in IN$				2	3	4	5	
	$h^{\#1}$	$x^{\#1}$	$pi^{\#1}$	$s1^{\#1}$	$x^{\#2}$	$s1^{\#2}$	$x^{\#3}$	$s1^{\#3}$	
h	$\Delta h^{\#1}$								
x	$\Delta h^{\#1}$	$\Delta x^{\#1}$			$\Delta x^{\#2}$				
pi			$\Delta pi^{\#1}$						
s1				$\Delta s1^{\#1}$					

3: $\boldsymbol{s1 = s1 + 4.0} * \sin(\boldsymbol{x} * \boldsymbol{pi})$

→(1). $\boldsymbol{EPM_{s1}} = \frac{\partial s1}{\partial s1^{\#1}}\boldsymbol{EPM_{s1}} + \frac{\partial s1}{\partial x}\boldsymbol{EPM_x} + \frac{\partial s1}{\partial pi}\boldsymbol{EPM_{pi}}$

$= \boldsymbol{EPM_{s1}} + 4 * \cos(x * pi) * (pi * \boldsymbol{EPM_x} + x * \boldsymbol{EPM_{pi}})$

→(2). $\boldsymbol{EPM}_{s1,s1^{\#2}} += tcErr(s1 + 4.0 * \sin(x * pi)$

	$iv \in IN$				2	3	4	5	
	$h^{\#1}$	$x^{\#1}$	$pi^{\#1}$	$s1^{\#1}$	$x^{\#2}$	$s1^{\#2}$	$x^{\#3}$	$s1^{\#3}$	
h	$\Delta h^{\#1}$								
x	$\Delta h^{\#1}$	$\Delta x^{\#1}$			$\Delta x^{\#2}$				
pi			$\Delta pi^{\#1}$						
s1	b1	b2	b3	b4	b5	$\Delta s1^{\#2}$			

4: $\boldsymbol{x = x + h}$

	$iv \in IN$				2	3	4	5	
	$h^{\#1}$	$x^{\#1}$	$pi^{\#1}$	$s1^{\#1}$	$x^{\#2}$	$s1^{\#2}$	$x^{\#3}$	$s1^{\#3}$	
h	$\Delta h^{\#1}$								
x	$2\Delta h^{\#1}$	$\Delta x^{\#1}$			$\Delta x^{\#2}$		$\Delta x^{\#3}$		
pi			$\Delta pi^{\#1}$						
s1	b1	b2	b3	b4	b5	$\Delta s1^{\#2}$			

5: $\boldsymbol{s1 = s1 + 2.0} * \sin(\boldsymbol{x} * \boldsymbol{pi})$

	$iv \in IN$				2	3	4	5	
	$h^{\#1}$	$x^{\#1}$	$pi^{\#1}$	$s1^{\#1}$	$x^{\#2}$	$s1^{\#2}$	$x^{\#3}$	$s1^{\#3}$	
h	$\Delta h^{\#1}$								
x	$2\Delta h^{\#1}$	$\Delta x^{\#1}$			$\Delta x^{\#2}$		$\Delta x^{\#3}$		
pi			$\Delta pi^{\#1}$						
s1	d1	d2	d3	b4	d5	$\Delta s1^{\#2}$		$\Delta s1^{\#3}$	

Aggregating columns according to variables :

after 2: $\boldsymbol{x = x + h}$

	h	x	pi	s1
h	$\Delta h^{\#1}$			
x	$\Delta h^{\#1}$	$\Delta x^{\#1}+\Delta x^{\#2}$		
pi			$\Delta pi^{\#1}$	
s1				$\Delta s1^{\#1}$

After the 5th assignments:

	h	x	pi	s1
h	$\Delta h^{\#1}$			
x	$2\Delta h^{\#1}$	$\Delta x^{\#1}+\Delta x^{\#2}+\Delta x^{\#3}$		
pi			$\Delta pi^{\#1}$	
s1	d1	d2+d5	d3	$b4+\Delta s1^{\#2} + \Delta s1^{\#3}$

Figure 2: The evolution of the EPM during the first iteration of Simpson's method. The EPM consists of four rows, each corresponding to one of four scalar variables. Since all four variables serve as input parameters to the function, a total of eight error sources must be considered: four within the loop body and four at the function entry. Consequently, the first eight columns of the EPM are elaborated in detail, with each column labeled according to the variable modified by the associated assignment statement. The superscript $\#i$ attached to these variables indicates that it is the i^{th} assignment to that variable. Each entry in the EPM is color-coded—white indicates a zero value, while green denotes a non-zero value. Additionally, notations are employed in non-zero entries to facilitate the identification of relationships among EPM updates or between EPMs with and without column compression. Following column compression, the EPM is reduced to four columns, and its states after Statement 2 and Statement 5 are illustrated at the bottom of the figure.

In the naive Algorithm 1, each entry of EPM represents the error sensitivity of an individual error source. After applying just-in-time accumulation, each entry represents the error sensitivity of an error group. Let $G(es)$ denote the user-provided grouping method that maps each error source es to its corresponding error group. The second sub-step update of the Algorithm 1 should be

$$\boldsymbol{EPM}_{r,G(t_k)} += tcErr(r). \tag{7}$$

Grouping methods for error sources. Basically, users can group assignments based on their result operands. For instance, for a scalar variable v, all assignments to the scalar variable v can be grouped into a single error group E_v. Similarly, for an array variable V, it is common practice to group all assignments to different array elements V_i into a unified error group E_V. Secondly, users can further group assignments along the time dimension, with the granularity of functions or loop iterations. During the lifetime of a variable, its assignments can bind to different error groups. A typical example involves splitting a loop into several subspaces, where assignments within different subspaces can be mapped to different error groups. With the help of compiler, this process imposes no additional burden on the programmer. Thirdly, the greedy strategies utilized during the mixed-precision allocation phase may lead to performance degradation by introducing excessive floating-point (FP) type conversions. To mitigate the overhead from these additional conversions, users can group multiple arrays or scalar variables into a single group based on the degree of intimacy among them, as determined by tools such as HiFPTuner [12].

Figure 2 also shows what EPM looks like when using column grouping at the bottom of the figure. Condensed EPM is shown after the execution of statement 2, and also after statement 5. For the latter case, we can see that EPM$_{s1,s1}$ is the summation of the three related entries of EPM$_{s1}$ under the naive version.

Validity of column compression. We need to prove that just-in-time accumulation of EPM columns produces the same result as post-mortem column accumulation. Consider two different analysis procedures, one using a compressed EPM, denoted as EPM', and the other using an uncompressed EPM. Since the columns of EPM evolve independently, it suffices to prove the correctness of the compression process for two columns, as the conclusion can be generalized to more columns. Suppose that columns i and j in EPM are grouped into column k in EPM', we need to prove that at the exit of program segment P, $\boldsymbol{EPM}_{y,i}+\boldsymbol{EPM}_{y,j} = \boldsymbol{EPM'}_{y,k}$. We use mathematical induction on l, the length of the assignment statement sequence, to prove that after the EPM update for any assignment $t_l : r = f(a,b)$ in T, $\boldsymbol{EPM}_{r,i} + \boldsymbol{EPM}_{r,j} = \boldsymbol{EPM'}_{r,k}$ still hold. It is trivial that the property holds when $l = 1$. Suppose the property holds for $l \leq n$. Now we consider assignment $t_{n+1} : r = f(a,b)$, supposing $G(r)$ is neither i nor j,

$$\begin{aligned}\boldsymbol{EPM}_{r,i} + \boldsymbol{EPM}_{r,j} =& (\frac{\partial f}{\partial a}\boldsymbol{EPM}_{a,i} + \frac{\partial f}{\partial b}\boldsymbol{EPM}_{b,i})\\ &+(\frac{\partial f}{\partial a}\boldsymbol{EPM}_{a,j} + \frac{\partial f}{\partial b}\boldsymbol{EPM}_{b,j})\\ =&\frac{\partial f}{\partial a}(\boldsymbol{EPM}_{a,i} + \boldsymbol{EPM}_{a,j})\\ &+\frac{\partial f}{\partial b}(\boldsymbol{EPM}_{b,i} + \boldsymbol{EPM}_{b,j})\end{aligned}$$

then according to the inductive hypothesis, it yields

$$\boldsymbol{EPM}_{r,i} + \boldsymbol{EPM}_{r,j} = \frac{\partial f}{\partial a}\boldsymbol{EPM'}_{a,k} + \frac{\partial f}{\partial b}\boldsymbol{EPM'}_{b,k} = \boldsymbol{EPM'}_{r,k}$$

When $G(r)$ is either i or j, the proof is similar. So we prove the induction step and finish the proof.

After column compression, EPM has a size of $D * |E|$, where D represents the total number of FP scalar variables in the program and E is the set of error groups. So, the time complexity of the improved Algorithm 1 becomes $O(|E|*|T|)$, and the space complexity turns $O(|E| * D)$.

3.3 Improve the Accuracy of the Algorithm 1

The error model used in the previous sections is still Equation 4, which completely ignores dependencies among assignments.

A new error model to capture dependencies. We modify the truncation error formula in the above Algorithm 1 from formula 7 to

$$\boldsymbol{EPM}_{r,G(t_k)} += tcErr(r + \boldsymbol{EPM}_{r,G(t_k)}) \tag{8}$$

This truncation error formula is different from the previous one, which is $tcErr(r)$. The new argument of $tcErr$ captures the cumulative error influence from all prior assignments within the same error group.

In order to put the accuracy property of the above formula formally, we introduce several notations on mixed precision program versions. For a certain error source group E_i, we introduce $C(E_i)$ to represent the related mixed precision version of the program, in which only the assignments in E_i use lower precision while all other assignments use high precision. For the assignment $t_k : r = f(a,b)$ in P, $f(a,b)^{C(E_i)}$ and $r^{C(E_i)}$ denote the value of each expression under $C(E_i)$, respectively. However, it is important to note that if $t_k \in E_i$, a precision conversion occurs before assigning $f(a,b)^{C(E_i)}$ to $r^{C(E_i)}$, i.e.,

$$r^{C(E_i)} = f(a,b)^{C(E_i)} + tcErr(f(a,b)^{C(E_i)})$$

Otherwise $r^{C(E_i)} = f(a,b)^{C(E_i)}$.

Theorem 3.1. *When applying the improved algorithm 1 (with the new truncation error formula 8) to a given program segment, at any time during analysis, for the current assignment $t_k : r = f(a,b)$, two sub-steps of EPM update to EPM$_r$ are the first order Taylor approximation to the actual accumulated errors originated from all error sources to the current assignment. This means that, for any error source group E_i,*

After the whole-vector update with EPM$_r$, we have

$$\boldsymbol{EPM}_{r,E_i} = f(a,b)^{C(E_i)} - f(a,b) \tag{9}$$

After updating EPM$_{r,t_k}$ with truncation error,

$$\boldsymbol{EPM}_{r,E_i} = r^{C(E_i)} - r \tag{10}$$

The proof of the theorem runs as follows.

We prove the property using mathematical induction on the length of the assignment statement sequence.

As before, input variables are regarded as special assignments. When the length is 1, there are only two cases. In one case, $t_k : r = f(a,b)$ is the first FP assignment of program P, and neither operands nor the result brings errors, the above property holds apparently. In the second case, t_k is an input variable that belongs to E_i, and again the above property holds apparently. All other cases turn the problem to one whose length is larger than 1.

Induction condition: we assume that when length $L \le k$, the above properties always hold.

For a program with k+1 assignments, and $t_{k+1} : r = f(a, b)$, we have

$$f(a,b)^{C(E_i)} = f(a^{C(E_i)}, b^{C(E_i)})$$

According to the induction assumption,

$$a^{C(Ei)} = a + \boldsymbol{EPM}_{a,E_i}$$
$$b^{C(E_i)} = b + \boldsymbol{EPM}_{b,E_i}.$$

So,

$$f(a,b)^{C(E_i)} = f(a + \boldsymbol{EPM}_{a,E_i}, b + \boldsymbol{EPM}_{b,E_i})$$

For the above formula, we firstly apply first-order Taylor series approximation, then after the whole-vector EPM update of statement t_{k+1}, simplify it using the EPM_r equation in sub-step 1, the formula turns as follows.

$$\begin{aligned} f(a,b)^{C(E_i)} \approx& f(a,b) + \frac{\partial f}{\partial a} * \boldsymbol{EPM}_{a,E_i} + \frac{\partial f}{\partial b} * \boldsymbol{EPM}_{b,E_i} \\ =& f(a,b) + \boldsymbol{EPM}_{r,E_i} \end{aligned}$$

This proves that Equation 9 holds.

If $t_k \notin E_i$, Equation 10 holds apparently. When $t_k \in E_i$, before the second EPM update, according to Equation 9

$$\begin{aligned} r^{C(E_i)} - r =& f(a,b)^{C(E_i)} - f(a,b) + tcErr(f(a,b)^{C(E_i)}) \\ \approx& \boldsymbol{EPM}_{r,E_i} + tcErr(f(a,b) + \boldsymbol{EPM}_{r,E_i}) \end{aligned}$$

Now, the right side expression is exactly the new value of EPM_{r,E_i} after the second EPM update with formula 8. So, Equation 10 holds.

According to the above Theorem, this improved algorithm with formula 8 can significantly improve the accuracy of error estimation for each error group with negligible overhead.

4 Implementation and Optimizations

We implemented the proposed algorithm in a tool called Light-FP, which is comprised of a compiler built upon LLVM-13 to perform code transformation and a runtime library realizing error sensitivity analysis. The compiler generates codes for local derivatives, inserts runtime calls for EPM updates, and also applies an optimization for EPM indexing. The runtime system is in charge of EPM management and EPM updates.

4.1 Programming Interface

Light-FP can analyze the whole program or a specified code region, and the output statement should be marked. The user should register all the FP variables explicitly before their first assignment statement. To register a variable, the user should provide the address and a string label to designate the related error source group, and the runtime system will bind the variable to a certain EPM row and a certain EPM column. The user can register a variable several times during its lifetime, each time with a different string label, if he/she wants to discriminate the variable's error contribution in different code regions.

Two kinds of registering interfaces. The basic interface is element-by-element registration, in which users provide the address and the column label for each variable. This kind of API is general enough to apply to irregular data structures such as graphs. We also provide batched mode registration for FP arrays, in which users provide the base address of the array and the array size. Under the latter API, array elements of the same array are all bound to the same column, and adjacent array elements are bound to adjacent EPM rows.

4.2 Compiler Transformations

The compiler works on the un-optimized, compiler intermediate representation of the program, and has two main compiler passes. The error propagating pass generates codes for local derivatives, and inserts runtime calls for EPM initialization and EPM updates. And we also realized an adjoint indexing optimization, called EpmIdxTrace.

Error propagating pass. In each user-specified target function, local derivative codes are generated for each FP assignment and runtime calls for EPM update are inserted. EV propagation is maintained on procedural boundaries, by pushing EVs to stack that correspond to FP arguments passed by value and popping stack the EV that is related to the function result. It should be noted that Light-FP does not update EPM at the granularity of float-pointing instructions, but at each source code level assignment, since allocating different precision for each virtual register is usually not necessary, but just adds more EPM columns and more type conversions.

EpmIdxTrace pass. The basic method of EPM indexing is through hashmap, taking the memory address of variables as keys. Although hashmap is usually highly optimized, but frequent hash computation and hashmap data access (bad cache locality) at every FP assignment, bring remarkable performance reduction. We note that under the batched mode of FP array registration, variables that have adjacent addresses have adjacent EPM row indices, so array elements of the same array can share one single table look-up. EpmIdxTrace pass tries to substitute hashmap retrieval calls with integer arithmetic, accompanied by FP pointer arithmetic of the original program. Firstly, it regards FP pointers of batch mode variable registration calls as seed variables, collects all the reachable FP pointer assignments (including GEP and bitcast, etc.) from them, and allocates two shadow variables (or adjoint variables) for each of the target pointers to store the related EPM row index and column index, respectively. Secondly, for each assignment of these target pointers, index propagation instructions among adjoint variables are generated, where copy between pointers corresponds to copies between adjoint indices, and pointer arithmetic corresponds to an offset operation on the adjoint row index accompanied with a copy of the adjoint column index. This allows us to access the EPM using the adjoint indices instead of relying on a hash map. If propagation fails (interrupted by an external function), the dynamic hash map retrieval is called. Thirdly, the compiler handles scalar stack variables in batched mode, assigning their relative positions statically, thus reducing register pressure incurred by adjoint variables. Fourthly, to support FP pointers in arbitrary nested aggregated structures, each targeted FP pointer is upgraded to a structure {double*, int32, int32}, and any aggregated structure including targeted FP pointers is also upgraded.

4.3 Runtime Support

The runtime system is designed for EPM management and update.

Preallocated EPM space. To lower the overhead of dynamic memory allocation, we use profiling to get the shape of the EPM matrix.

EPM binding and indexing. We use the hash map to retrieve the EPM index for each variable, where the address of the variable is the key, and the value contains both the row index and the column index. Since each variable has a lifetime, so multiple variables may share the same address (key of the hashmap) during the program execution. In order to avoid memory fragmentation, we do not reclaim the EPM row when a variable becomes inactive but handle such a problem with a conflict table.

Data parallelization on EPM update. The primary overhead in the proposed algorithm arises from the EPM update. However, the independent nature of EPM update across distinct error source groups provides substantial opportunities for data parallelism. The degree of data parallelism is EPM columns, when it is wider than the SIMD width of the target architecture, thread parallelism is introduced. We adopt the master-helper pattern to realize thread parallelism: the master thread executes the analyzed code region, evaluates derivatives, and assigns EPM update tasks to the downstream helpers, while helper threads receive the tasks, and perform the related EPM update using SIMD instructions. Efficient communication between the master and helper threads is achieved through double buffering. The workload of the helper threads is evenly distributed according to EPM columns, with each thread operating exclusively on its designated columns. To minimize cache thrashing and improve cache locality, the EPM matrix is stored in column tiles.

5 Evaluation

5.1 Methodology and Experimental Setup

We compared Light-FP against the current state-of-the-art CHEF-FP using eight different benchmarks. We chose to run both CHEF-FP and Light-FP in docker container with privileged mode throughout the evaluation section. We used clang++-13 as the backend compiler in the experiments, and all of the tests used the -O3 option. We ran each experiment five times and got the data on average. The time taken was measured using std::chrono library and peak memory by GNU time. The experiments were done on a Dell PowerEdge R750 server with two 2.2GHz, Intel Xeon Gold 5320 processors, each processor has 26 cores. The server has 39MB L3 cache and 128 GB of memory, and we disabled hyperthreading in the experiments.

HPC-MixPBench[24] is an HPC benchmark suite for mixed-precision analysis, consisting of a representative set of kernels and applications that are widely used in the HPC domain. We chose six out of seven applications from it: Black-Scholes, Hotspot, HPCCG, LavaMD, K-means, and SRAD. Additionally, we chose NPB/CG from the NAS parallel benchmarks [26], and Simpsons which is usually used in the error analysis area.

5.2 The Accuracy of Error Estimation on Single Variables

We compare the estimation error of Light-FP and CHEF-FP with the actual error in Table 1 to illustrate the accuracy of Light-FP, by demoting the precision of one variable at a time. CHEF-FP actually adopts formula 6 as the error model. To reveal the deep reason for the current method's inaccuracy, we present two kinds of estimated errors for CHEF-FP, the modified CHEF-FP (without *abs()* treatment, i.e. using error model 4) and the original CHEF-FP (with *abs()* treatment).

Due to space limitations, we focus on variables with significant output errors as these variables will have a greater impact on precision allocation.

In Table 1, to highlight the deviation from the actual errors, the blue color is used when the deviation is about one order of magnitude, red when the difference is above one order of magnitude, and the order difference is given in parentheses. As shown, Light-FP's estimation is very close to the actual error, only 4 variables are marked blue. Modified CHEF-FP (without *abs*) generally significantly underestimates the output error, which is consistent with our observation in Simpsons in section 2.1. For the original CHEF-FP, the influence of *abs()* processing is huge, making the estimations 3 to 6 orders of magnitude larger. But CHEF-FP's estimation is still

Table 1: Comparison between the estimated error and the actual error. CHEF-FP has two columns, one without using *abs()* when accumulation and one enable *abs()*, denoted as *Esti-CHEF-w/o ABS* and *Esti-CHEF-w/t ABS*, respectively. The estimated error of Light-FP is denoted as *Esti-Light*.

Application	Input	Variable	Actual Error	Esti-Light	Esti-CHEF-w/o ABS	Esti-CHEF-w/t ABS
Simpsons	10^6	s1	-2.64E-03	-2.64E-03	-8.16E-05(-2)	3.93E-02(+1)
		x	-3.48E-02	-3.51E-02	-2.39E-07(-5)	4.21E-02
lavaMD	8	fv_cpu_v	3.35E-04	3.35E-04	1.63E-07(-3)	2.60E-02(+2)
		fv_cpu_x	-6.10E-06	-6.10E-06	1.22E-07(-1)	1.24E-02(+4)
		fv_cpu_y	7.34E-07	7.37E-07	-1.16E-06(+1)	1.24E-02(+5)
		fv_cpu_z	-2.67E-06	-2.67E-06	-1.44E-08(-2)	1.24E-02(+4)
hotspot	2^{20}	norm	2.14E+00	2.14E+00	-3.33E-04(-4)	3.81E+00
		temp	-1.02E-04	-9.99E-05	-1.55E-08(-4)	7.63E-06(-2)
		result	-1.02E-04	-1.02E-04	1.41E-08-4	7.62E-06(-2)
BlackScholes	10^6	OutputX	-1.71E-01	-1.71E-01	-2.34E-02(-1)	3.01E-01
		xK2	-3.75E-01	-3.40E-01	-2.69E-01	5.98E+00(+1)
		xLocal	-2.37E-01	-2.37E-01	-2.27E-01	2.83E+00(+1)
		inv_sqrt_2xPI	3.88E-01	3.88E-01	3.88E-01	4.72E-01
NPB/CG	14000	zeta	9.06E-07	9.06E-07	-4.74E-08(-1)	4.74E-08(-1)
		norm_temp1	6.52E-06	6.52E-06	1.28E-06	4.79E-04(+2)
		norm_temp2	5.28E-07	1.01E-06(+1)	-4.32E-06(+1)	4.25E-04(+3)
HPCCG	80	x	-5.96E-07	2.38E-07	-8.97E-08	1.15E-06(+1)
		r	-5.54E-08	-2.20E-08	-4.68E-08	1.22E-07(+1)
		Ap	-2.79E-08	-2.18E-09(-1)	0.00E+00	0.00E+00
SRAD	500	Jc	-1.38E-07	1.13E-07	0.00E+00	0.00E+00
		image	-4.40E-06	-1.07E-06	2.19E-07(-1)	2.75E-03(+3)
K-means	10^6	ans	1.38E-03	4.90E-02(+1)	-1.08E-03	4.74E+01(+4)
		clusters	-4.31E-04	6.22E-03(+1)	-2.87E-03(+1)	3.17E-01(+3)

Table 2: Error estimation and performance of the chosen mixed precision configurations.

Benchmark	Input	Threshold	Actual Error	Estimated Error	Speedup
Simpsons	10^6	1e-06	6.62e-08	7.14e-08	1.72
hotspot	2^{18}	1e-08	-1.43e-10	-1.74e-10	1.09
NPB/CG	7000	1e-07	-3.12e-08	4.28e-09	1.04
HPCCG	20*30*80	1e-07	-2.27e-08	-2.19e-08	1.04
lavaMD	8	1e-05	-1.58e-06	-3.29e-06	1.71
SRAD	502*408(500)	1e-05	-4.42e-06	-1.30e-06	1.42

far from accurate, with a lot of red items and blue items, actually, it gives a serious overestimation of the output error.

The four inaccurate items of Light-FP that occurred in NPB/CG, HPCCH, and K-means are caused by certain nonlinear operators, such as division, *sqrt()*, and *exp()*. The second-order derivative of those operators may be much larger than the first derivative, which means the second-order terms in the Taylor series expansion cannot be neglected. Introducing second-order terms to the error model could address this issue but would incur substantial computational and memory overhead.

5.3 Evaluation of Mixed Precision Results

With the error sensitivity vector, we use the greedy algorithm[21] to find the optimal mixed precision configuration that satisfies the specified accuracy.

For each input of the tested benchmark, We applied error analysis and used HiFPTuner as a preprocessor to group the related variable, then searched for optimal mixed precision configurations based on the result of analysis, and chose one configuration for each benchmark with the best speedup, and showed them in Table 2. In the table, Light-FP's estimation is compared with the actual error, and speedups of the mixed-precision configuration are also given. One configuration that exhibits significant speedup under one input may not sustain the same acceleration in another input. This is due to variations in data size, computational complexity, and precision requirements at different scales can lead to increased overhead, resulting in performance degradation. We found that preprocessing by HiFPTuner is useful for Hotspot and NPB/CG.

5.4 Evaluation of Two Optimizations in Light-FP

The two performance optimizations that we described in subsection 4.2 and 4.3 are EpmIdxTrace and data parallelization on EPM update and are abbreviated as EIT and DP, respectively. Totally we should consider four combinations, no-opt (serial-analysis), EIT-only, DP-only, and EIT+DP. Since EIT-only is not important , so we omit it for page limits.

In the data parallelization scenario, both mismatched workload (a helper usually has more workload than the master) and inter-thread synchronization between the master and helpers may affect the pipelining performance apparently. To exclude the dragging back effects from helpers, we introduce two master versions, DP-master and DP-master+EIT. In these versions, both EPM update and helper threads are disabled, but the master still performs message packaging and data copying to the double buffers. In DP-master, hash map retrieval is used for EPM indexing, while EIT is enabled in DP-master+EIT. These versions imply the upper bounds of data parallelization when EIT is disabled/enabled. Besides, there are also two full versions, which are DP and DP+EIT. DP+EIT uses 8 helper threads for error updates, while DP uses 4 helper threads which is the optimal configuration.

Figure 3 is a combo chart, the primary Y-axis is normalized time according to the un-optimized version which is the serial analysis with the two optimizations disabled, and the secondary Y-axis shows speedup. The number of EPM columns is labeled below each benchmark name.

Contribution of EpmIdxTrace: From Figure 3, we can see positive EIT improvements both on the master side and on the overall performance in almost all cases. DP-master of Black-Scholes is the only exception, EIT shows no speedup because hashmap has only 44 items (good cache locality) and dynamic retrieval behaves almost as well as EIT. Furthermore, if *EpmIdxTrace* does not let multiple FP scalars share the same adjoint indices, EIT will witness severe performance degradation for high register pressure. We can also see that EIT is more profitable for large inputs than small inputs. For the NPB/CG case, EIT is 15% more effective on a large input compared to a small input. Table 3 presents the effect of EIT optimization for each application. EIT-imp1 gives the performance improvement of *EpmIdxTrace* on the master side with EPM update disabled, and EIT-imp2 relates to the overall performance contribution when using 8 helper threads. Among all cases, EIT-imp1 gets 41.88% improvement on average, and EIT-imp2 gains only 24.38%.

5.5 Performance Evaluation

Figure 4 to 11 compare the performance of Light-FP, CHEF-FP with that of the original program, both memory requirement and execution time will be considered

Accumulated effects of the two optimizations: The line chart in Figure 3 gives the overall speedup of DP+EIT over the un-optimized version. The average speedup among all 16 cases reaches 3.77.

Scalability of data parallelization: The positive improvement of EIT+DP over DP indicates that the master is the bottleneck in DP. But when EIT is enabled, things are different, helper threads become the bottleneck in many cases. The reason may lie in that the EPM update becomes memory-bound, and more helpers worsen the memory contention. For cases with EPM columns within 10-20, the proper number of helper threads is 6, and 8 threads are needed when columns are larger.

Memory Advantage: The memory bloat factor is constant for each benchmark under Light-FP, regardless of inputs or loop counts, and is quite close to the columns of its EPM. The bloat factor is sometimes smaller than *ncols* because only a small portion of the program's variables are monitored in the analysis. In contrast, the memory requirement of CHEF-FP grows with the number of loop counts.

Light-FP demonstrates evident memory advantages in cases with a large number of iterations. In NPB/CG with 14,000 iterations, the bloat factor of CHEF-FP is 942, while that of Light-FP is less than 2. In lavaMD with a size dimension of 10, CHEF-FP bloated 6,671x, whereas Light-FP bloated 32x. In the Simpsons case, Light-FP is 2,600X more memory efficient than CHEF-FP. In Hotspot, SRAD, and HPCCG,

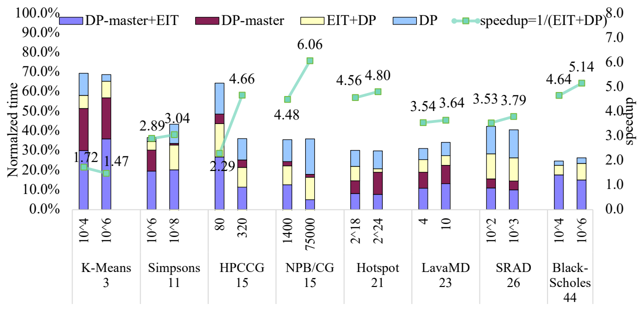


Figure 3: Effects of the two optimizations under the data parallelization scenario

Table 3: Effects of the EIT optimization.

	K-means	Simpsons	HPCCG	NPB/CG	Hotspot	lavaMD	SRAD	Black-Scholes	Average
EIT-imp1	40%	40%	53%	72%	59%	41%	30%	0%	41.88%
EIT-imp2	9%	22%	29%	50%	30%	20%	35%	0%	24.38%

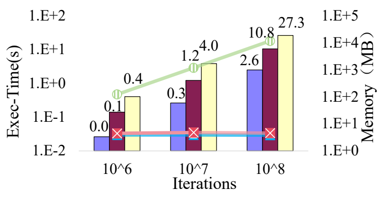


Figure 4: Simpsons

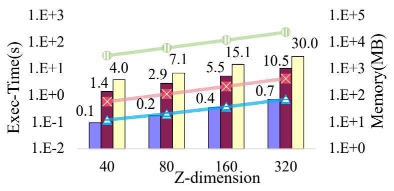


Figure 5: HPCCG

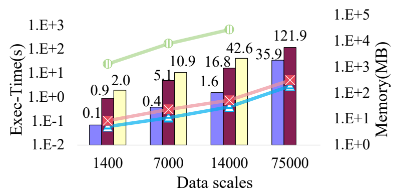


Figure 6: NPB/CG

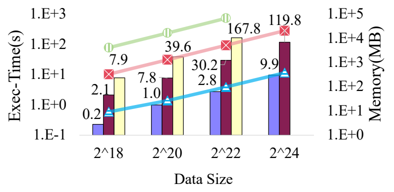


Figure 7: Hotspot

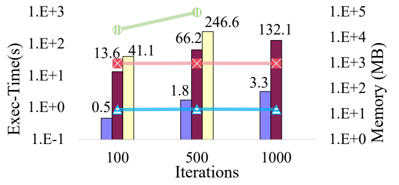


Figure 8: SRAD

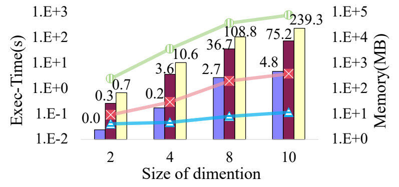


Figure 9: LavaMD

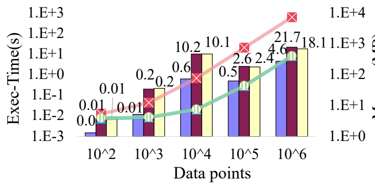


Figure 10: K-means

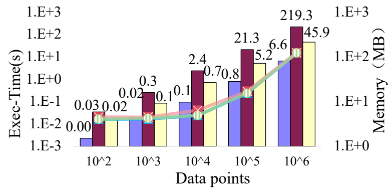


Figure 11: Black-Scholes

Figures 4 to 11 compare the performance of Light-FP, CHEF-FP with the original program. The bars represent the time taken by the tools, and the lines show peak memory usage. Both Y-axes use a logarithmic scale.

Light-FP exhibits 12x to 54x more efficient than CHEF-FP. Additionally, for the largest inputs tested in the NPB/CG and Hotspot cases, CHEF-FP encounters out-of-memory issues.

There are exceptions in two cases. For K-means and Black-Scholes, Light-FP's bloat factor on memory usage is only 1.3x~4x, but CHEF-FP performs perfectly with negligible memory usage. The reason is that for these two programs, CHEF-FP only focuses on the kernel function within a loop, and each execution of the kernel is independent of other executions. So CHEF-FP can aggregate the error and then release all the memory at the end of the execution of each kernel.

Time Advantage: Light-FP is obviously faster than CHEF-FP by about 2x~5x in six benchmarks, which are Hotspot, lavaMD, SRAD, Simpsons, NPB/CG, and HPCCG.

For the largest input tested of Hotspot, SRAD, and NPB/CG, CHEF-FP collapses for lacking memory, so their time bars are absent. For two of them, Hotspot and SRAD, we employed linear fitting to predict the running time of CHEF-FP on a machine with sufficient memory, as shown in Table 4. The predicted time can be considered as the minimum possible time since checkpointing introduces additional computational overhead. Nevertheless, Light-FP still demonstrates a runtime advantage in this extreme scenario (1.93x for Hotspot and 3.42x for SRAD). For NPB/CG, the running time at the maximum scale of 75,000 increases significantly due to a substantial rise in the number of iterations, which does not scale linearly, so the forecast time is not provided.

Table 4: Time comparison of LightFP with CHEF-FP whose time is gained by linear fitting.

Benchmark	Light-FP(s)	CHEF-FP(s)
Hotspot(2^{24})	119.8	231.6
SRAD(1000)	132.1	452.2

There are also exceptions for K-means and Black-Scholes. For most inputs of K-means, the analysis time of Light-FP is about 1.06x slower. The reason lies in that, Light-FP has about triple the instructions of CHEF-FP, and the 3-length error vectors exhibit poorly either in SIMD parallelism or in thread-level parallelism. For Black-Scholes, Light-FP's EV length is quite large (44), so the computation complexity is about 14 times larger than that of CHEF-FP. Although it gained 5.1x speedup from data parallelism, the runtime overhead is still large.

In summary, Light-FP exhibits an obvious advantage in memory consumption with up to 2600x more memory efficiency and also runtime advantages on many applications. For multi-entrant code regions with very few instructions, Light-FP may be less efficient than CHEF-FP in terms of memory consumption and runtime, but the difference is minor.

6 Current limitations

Input dependent: Dynamic analysis is input-dependent. The mixed precision configurations stemming from the error sensitivity result under one input may not apply to another input. Representative datasets should be used to get a general mixed-precision configuration for the application.

Support external functions: Currently Light-FP does not support external functions. In the future, third-party AD tools can be leveraged to generate code for them to get the related derivatives and enable the propagation of errors in the external function.

Support parallel programs: Since our algorithm adopts a tightly coupled approach combining AD with error computation, but not relying on third-party AD tools, the proposed algorithm should be tightly integrated with the implementation of each parallel programming language. For OpenMP, since EPM is a shared data structure among OpenMP threads, data synchronization on EPM is indispensable, and the synchronization of EVs is just an analog of that of variables in the original OpenMP program. For MPI programs, EV propagation among processes should be introduced along with MPI communication, and this means extra shadow communication on error vectors.

Improve the accuracy of error estimation: For some HPC applications, the first-order Taylor approximation is not accurate enough. The proposed algorithm can be further improved with a second-order error term as in [30], but not higher-order Taylor approximation because [22] revealed that higher-order Taylor approximation does not give much better approximation results, only leads to complex expressions. And the second-order error term can be kept in another EPM matrix.

7 Related Work

Floating-point error analysis has been studied extensively in the literature, and there are mainly three kinds of methods.

The static method analyzes the source code of the program, to get a tight bound on the FP rounding error which applies to all inputs. Some of them such as Rosa and Daisy [9], [7],[6] analyze real-valued programs with error specifications, and search for proper mixed precision configurations using SMT solver. Gappa[10] evaluated error bound for elementary functions using interval arithmetic and proved the result automatically, and its successor, Gappa++[20] utilized affine arithmetic in error bound evaluation and can further handle transcendental functions. Since the above techniques often provide very pessimistic overestimates, FPTaylor put forward Symbolic Taylor Expansions, a rigorous global optimization method, to tighten the error bound. Symbolic Taylor Expansions are also used in FPTuner[5] to get a mixed precision configuration that satisfies output accuracy by solving a quadratically constrained quadratic program. Static analysis is currently limited to kernels or small programs, and cannot scale to real applications.

Search-based methods are more general, apply to arbitrary programs theoretically, and can usually find more efficient, mixed-precision configurations. Precimonious introduced Delta-Debugging to search for a feasible configuration and leverage automatic code transformation to generate program variants. CRAFT[17] used binary instrumentation to build mixed-precision variants of existing binaries and a simple, breadth-first search for a proper mixed-precision configuration. Since fewer type casts at runtime lead to better performance, HiFPTuner[13] formulated a community hierarchy on variables based on edge profiling and dependence analysis and presented a hierarchical search algorithm that iteratively lowers precision with regards to communities. [3] built a backtrace graph based on control flow information and temporal locality, applied backtrace clustering (community) and iterative search refinement to reduce search complexity. GPUMixer[15] chose candidate assignment statements for mixed precision computation using a performance-driven approach and estimated errors through shadowed execution.

[19] modeled the actual performance and power consumption of mixed precision programs on hardware platforms, and based on this model searched for the Pareto optimal set of all precision configurations. Although the search-based approach is general, the state space is exponential to the number of variables so its scalability is limited, and it can also get trapped into local minima.

Compared with time-intensive, search-based approaches, dynamic analysis has low overhead, and the runtime information collected may be more relevant for finding a better solution. Blame[23] built the blame set for each instruction with the help of shadowed execution, and deduced precision requirements for each instruction, so reducing the state space to be explored dramaticallyADAPT suggested implementing an error sensitivity profile based on algorithmic differentiation, demonstrating its usefulness to narrow the search space, and has been incorporated with several software tools into an integrated toolchain, where TypeForge analyzes variable types and transforms codes, ADAPT provides error sensitivity information, and CRAFT searches the state space that has been reduced by previous tools. To deal with the performance problem of ADAPT, CHEF-FP chose Clad[31] as the AD tool, generated FP error estimation code directly into the derivative source, enabling compiler optimizations resulting in less analysis time and reduced memory usage. But, CHEF-FP does not completely solve the memory overhead problem, incurred by reverse mode AD, where space overhead is proportional to the total number of instructions of the original program in the worst case.

In contrast, Light-FP is built upon forward mode AD, with the help of EPM and error source aggregation, overcoming the time complexity issue of forward AD without hurting the accuracy of error analysis. This method is more suitable for HPC workloads.

8 Conclusion

In this paper, we propose an efficient algorithm to evaluate approximation errors, which tightly integrates forward mode AD with error source aggregation. we provide a new error model to capture correlation among variables within the same error source group. Potentially, the approach works for different parallel codes, including OpenMP, MPI, and CUDA.

We realized the algorithm in Light-FP, a source-code transformation tool, to analyze approximation errors in HPC applications, and evaluated it on six applications and two other benchmarks. We find that Light-FP can provide more accurate error estimation than traditional methods, and show how the estimation is used in mixed-precision tuning. Low overhead makes Light-FP feasible for typical HPC applications with a large number of iterations. Light-FP achieved a maximum speedup of 5.6X over CHEF-FP in the Hotspot application. It also significantly reduced memory by 2600X for the Simpsons benchmark, 500X for CG, and 200X for LavaMD.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments and suggestions. In addition, we used ChatGPT to polish some sentences in this paper and write some Python scripts. This work is supported by the fund of Laboratory for Advanced Computing and Intelligence Engineering.

References

[1] Alan Ayala, Stanimire Tomov, Xi Luo, Hejer Shaeik, Azzam Haidar, George Bosilca, and Jack Dongarra. 2019. Impacts of multi-gpu mpi collective communications on large fft computation. In *2019 IEEE/ACM Workshop on Exascale MPI (ExaMPI)*. IEEE, 12–18.

[2] Marc Baboulin, Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Julie Langou, Julien Langou, Piotr Luszczek, and Stanimire Tomov. 2009. Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications* 180, 12 (2009), 2526–2533.

[3] Hugo Brunie, Costin Iancu, Khaled Z Ibrahim, Philip Brisk, and Brandon Cook. 2020. Tuning floating-point precision using dynamic program information and temporal locality. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[4] Yohan Chatelain, Eric Petit, Pablo de Oliveira Castro, Ghislain Lartigue, and David Defour. 2019. Automatic exploration of reduced floating-point representations in iterative methods. In *Euro-Par 2019: Parallel Processing: 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26–30, 2019, Proceedings 25*. Springer, 481–494.

[5] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2017. Rigorous floating-point mixed-precision tuning. *ACM SIGPLAN Notices* 52, 1 (2017), 300–315.

[6] Eva Darulova, Einar Horn, and Saksham Sharma. 2018. Sound mixed-precision optimization with rewriting. In 2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS).

[7] Eva Darulova, Anastasiia Izycheva, Fariha Nasir, Fabian Ritter, Heiko Becker, and Robert Bastian. 2018. Daisy-framework for analysis and optimization of numerical programs (tool paper). In *Tools and Algorithms for the Construction and Analysis of Systems: 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I 24*. Springer, 270–287.

[8] Eva Darulova and Viktor Kuncak. 2014. Sound compilation of reals. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 235–248.

[9] Eva Darulova and Viktor Kuncak. 2017. Towards a compiler for reals. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 39, 2 (2017), 1–28.

[10] Florent De Dinechin, Christoph Quirin Lauter, and Guillaume Melquiond. 2006. Assisted verification of elementary functions using Gappa. In *Proceedings of the 2006 ACM symposium on Applied computing*. 1318–1322.

[11] Andreas Griewank. 1992. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and software* 1, 1 (1992), 35–54.

[12] Hui Guo and Cindy Rubio-González. 2018. Exploiting community structure for floating-point precision tuning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 333–343.

[13] Hui Guo and Cindy Rubio-González. 2018. Exploiting community structure for floating-point precision tuning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 333–343.

[14] Laurent Hascoet and Valérie Pascual. 2013. The Tapenade automatic differentiation tool: principles, model, and specification. *ACM Transactions on Mathematical Software (TOMS)* 39, 3 (2013), 1–43.

[15] Ignacio Laguna, Paul C Wood, Ranvijay Singh, and Saurabh Bagchi. 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 34*. Springer, 227–246.

[16] Michael O Lam and Jeffrey K Hollingsworth. 2018. Fine-grained floating-point precision analysis. *The International Journal of High Performance Computing Applications* 32, 2 (2018), 231–245.

[17] Michael O Lam, Jeffrey K Hollingsworth, Bronis R de Supinski, and Matthew P Legendre. 2013. Automatically adapting programs for mixed-precision floating-point computation. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*. 369–378.

[18] Michael O Lam, Tristan Vanderbruggen, Harshitha Menon, and Markus Schordan. 2019. Tool integration for source-level mixed precision. In *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE, 27–35.

[19] Zeqing Li, Yongwei Wu, and Youhui Zhang. 2023. Multi-Objective Optimization for Floating Point Mix-Precision Tuning. In *2023 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 1–6.

[20] Michael D Linderman, Matthew Ho, David L Dill, Teresa H Meng, and Garry P Nolan. 2010. Towards program optimization through automated analysis of numerical precision. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. 230–237.

[21] Harshitha Menon, Michael O Lam, Daniel Osei-Kuffuor, Markus Schordan, Scott Lloyd, Kathryn Mohror, and Jeffrey Hittinger. 2018. Adapt: Algorithmic differentiation applied to floating-point precision tuning. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 614–626.

[22] Arnold Neumaier. 2003. Taylor forms—use and limits. *Reliable computing* 9, 1 (2003), 43–79.

[23] Cuong Nguyen. 2015. A Dynamic Analysis for Tuning Floating-point Precision. (2015).

[24] Konstantinos Parasyris, Ignacio Laguna, Harshitha Menon, Markus Schordan, Daniel Osei-Kuffuor, Giorgis Georgakoudis, Michael O Lam, and Tristan Vanderbruggen. 2020. HPC-MixPBench: An HPC benchmark suite for mixed-precision analysis. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 25–36.

[25] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H Bailey, Costin Iancu, and David Hough. 2013. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*. 1–12.

[26] William Saphir, Rob F Van der Wijngaart, Alex Woo, and Maurice Yarrow. 1997. New Implementations and Results for the NAS Parallel Benchmarks 2.. In *PPSC*. Citeseer.

[27] Garima Singh, Baidyanath Kundu, Harshitha Menon, Alexander Penev, David J Lange, and Vassil Vassilev. 2023. Fast And Automatic Floating Point Error Analysis With CHEF-FP. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1018–1028.

[28] Jeffrey Mark Siskind and Barak A Pearlmutter. 2018. Divide-and-conquer checkpointing for arbitrary programs with no user annotation. *Optimization Methods and Software* 33, 4-6 (2018), 1288–1330.

[29] Alexey Solovyev, Marek S Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. 2018. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 41, 1 (2018), 1–39.

[30] Alexey Solovyev, Marek S Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. 2018. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 41, 1 (2018), 1–39.

[31] Vassilev Vassilev, M Vassilev, A Penev, L Moneta, and V Ilieva. 2015. Clad—automatic differentiation using clang and LLVM. In *Journal of Physics: Conference Series*, Vol. 608. IOP Publishing, 012055.