使用 CHEF-浮点进行快速自动浮点误差分析

Garima Singh*§，Baidyanath Kundu*§、Harshitha Menon†、Alexander Penev‡、David J. Lange§、Vassil Vassilev§*

*欧洲核子研究组织，*Espl.desParticules 1*，*1211* 梅兰，瑞士§物理系，普林斯顿大学，普林斯顿，新泽西州 *08544*，美国
†劳伦斯利弗莫尔国家实验室应用科学计算中心，美国加利福尼亚州利弗莫尔 *94551*
‡数学与信息学学院，普罗夫迪夫大学，保加利亚大道*236*号，*4000* 普罗夫迪夫，保加利亚 电子邮件：*{*garima.singh, baidyanath.kundu, vassil.vassilev*}*@cern.ch, †*harshitha@llnl.gov*, ‡*apenev@uni-plovdiv.bg*,* § *david.lange@princeton.edu*

arXiv:2304.06441v1 [math.NA] 13 Apr 2023

摘要——**随着我们逼近摩尔定律的极限，研究人员正在探索不同的范式，以实现前所未有的性能。近似计算（AC）依赖于应用程序能够容忍结果中的一定误差，以牺牲准确性换取性能，已展现出巨大潜力。尽管 AC 在机器学习等领域取得了成功，但由于高性能计算（HPC）对准确性要求严格，其在该领域的应用仍有限。我们需要能够识别代码中适合近似处理的区域，并评估其对应用程序输出质量影响的工具和技术，从而指导开发者有选择地采用近似计算。为此，我们提出 CHEF-FP：一种灵活、可扩展且易于使用的源代码转换工具，它基于自动微分（AD），用于分析高性能计算应用中的近似误差。**

CHEF-FP 使用 Clad 作为后端。Clad 是一款高效的自动微分工具，作为 Clang 编译器的插件构建，并基于 LLVM 编译器基础设施；它利用自身的自动微分能力来评估 C++ 代码中的近似误差。CHEF-FP 在源代码级通过向生成的伴随代码中注入误差估计代码来工作。这样，误差估计代码便可接受编译器优化，从而缩短分析时间并减少内存使用量。我们还对基于源代码转换的自动微分工具进行了理论和架构层面的扩展，使其能够执行浮点误差分析。本文主要关注分析混合精度近似计算技术引入的误差，这是高性能计算中最流行的近似技术。我们还通过在使用近似函数的代码上评估本工具，展示了它在估计其他类型误差方面的适用性。此外，我们展示了 CHEF-FP 在分析时间上的加速比，并将其与现有最先进工具进行了比较。这得益于它能够直接在导数源代码中生成并插入近似误差估计代码。生成的代码也更有望获得更好的编译器优化，从而降低运行时性能开销。

I. 引言

随着我们步入后摩尔时代，晶体管特征尺寸缩小不再能带来性能增长的免费午餐，研究人员正在探索其他计算范式，以提高计算吞吐量。近似计算（AC）作为一种提升峰值性能的有前景的方法，受到了广泛关注。近似计算依赖应用程序容忍一定程度的误差，从而实现性能提升。在现有的各种近似计算技术中，降低浮点精度（即混合精度）越来越受欢迎。计算机体系结构支持多种浮点数据和算术运算精度级别——64 位双精度、32 位单精度、128 位四精度和 16 位半精度。精度的选择决定了舍入误差的大小。对数据和运算采用更高精度虽可能提高准确性，但也会增加应用程序的执行时间、内存和能耗。混合精度调优是在必要时使用更高精度以保持准确性，并在能够提升性能的地方使用较低精度。

尽管浮点提供了多种精度级别，但在高性能计算和科学计算代码中应用这些精度颇具挑战。要有效地使用它们，开发人员需要了解舍入误差的细节，以及误差如何在应用程序中传播。由于缺乏可扩展且严谨的工具来分析应用程序代码区域的误差敏感度，开发人员往往会选择更稳妥的方案，即始终使用高精度。我们需要可扩展的工具，以了解降低数据和计算精度的影响，识别可容忍误差的区域，并为用户提供指导。

目前已有多种技术被提出，用于估算应用程序的敏感性分布，包括基于自动搜索的方法 [1], [2]，静态分析 [3], [4]，或使用自动微分（AD） [5]。遗憾的是，现有工具无法为高性能计算应用提供可行的解决方案。搜索方法的成本非常高，因为状态空间极大，即使是小型基准测试，也很快就会变得不可行。基于静态分析的方法采用区间分析或泰勒级数近似，能够严格估算浮点误差，但目前仅适用于操作数量较少的程序 [4]。有些方法 [5], [6] 利用自动微分来确定代码中对误差具有韧性的区域。虽然这些方法已被证明在较小规模的高性能计算基准测试中效果良好，但通常需要手动修改代码，并将多种软件工具整合在一起 [7]。这些工具速度慢且内存开销大，因此无法用于大型高性能计算工作负载。

©2012 IEEE。允许个人使用本材料。对于任何其他用途，包括在任何当前或未来媒体中重印/再版本材料以用于广告或宣传目的、创作新的汇编作品、转售或重新分发至服务器或列表，或在其他作品中重复使用本作品的任何受版权保护部分，均须获得 IEEE 的许可。

我们提出 CHEF-FP，这是一款可扩展、灵活且易于使用的工具，用于分析高性能计算应用中的近似误差。我们使用 Clad [8]，作为所提出框架的后端。Clad 是一款高效的自动微分工具，以插件形式构建于 Clang 编译器之上，并利用其自动微分功能来评估 C++代码中的浮点误差。Clad 通过转换程序的编译器内部表示来构建导数。这种粒度使作为 Clad 扩展构建的 CHEF-FP 能够利用程序的源代码信息，根据一组规则自动向生成的导数添加辅助指令。因此，CHEF-FP 无需用户干预即可为代码添加浮点误差信息。对于大型代码而言，这种自动化非常有益，因为手动编写注释会变得复杂、繁琐且容易出错。此外，借助基于编译器的后端更高的灵活性，CHEF-FP 能够进行特定领域的浮点误差分析。通过轻松修改 CHEF-FP 框架的部分内容，可以实现更加定制化的浮点误差分析。这种可变性支持更精细的分析，用户可以自行查询源代码信息，并针对分析方向做出更准确的决策。

最后，由于我们将浮点误差估算（EE）代码直接生成到导数源代码中，因此与执行类似浮点误差分析的其他工具相比，显著加快了速度。生成的代码也有望进一步进行编译器优化，从而提升运行时性能。我们使用 CHEF-FP 提供的浮点误差配置文件来指导代码混合精度版本的设计，在满足用户指定的误差阈值的同时，使*HPCCG* 的性能提升了 8%，布莱克-斯科尔斯 的性能提升了 65%。在分析阶段，CHEF-FP 在辛普森基准测试中实现了最高 217% 的时间加速比，在布莱克-斯科尔斯基准测试中实现了 632% 的内存效率。

主要贡献

- CHEF-FP 是一种高效工具，可自动执行基于自动微分的浮点误差分析。CHEF-FP 将误差计算内联到伴随代码中，使分析速度更快、内存效率更高，因此适用于数据密集型应用程序；
- 用于扩展自动微分以执行浮点误差分析的形式化方法。具体而言，我们提出了一种通用方法，用于扩展基于源代码转换的自动微分工具；
- 可定制的 EE 模块，支持任意基于 AD 的用户定义误差模型；
- 使用一组高性能计算基准测试，对工具进行评估，以分析混合精度和近似函数的误差。

II. 背景

浮点运算是高性能计算应用中最普遍的计算。计算机体系结构支持浮点数据和算术运算的多个精度级别。在 IEEE 754 标准 [9]，（如今实数最常见的表示形式）中，可选择 64 位双精度、32 位单精度、128 位四倍精度，以及16 位半 精度。精度的选择决定了舍入误差的大小。

浮点舍入误差是各变量产生的浮点误差在目标函数结果中的累积。目标函数被建模为一系列赋值操作。这些赋值操作和函数输入被假定为相互独立（针对相关变量的浮点误差分析仍处于起步阶段，尚未形成成熟的估算方法）。我们定义了一个表达式，用于计算并累积每次赋值引入的浮点误差，并将这些误差计算与累积表达式称为浮点误差模型。CHEF-FP 旨在估算这些浮点舍入误差的上界。

A. 浮点误差建模

误差模型描述了一种可用于误差估计分析的度量。该度量应用于函数中的所有赋值操作，其计算结果会累积到赋值左侧的总浮点误差中。泰勒级数近似非常适合用于对浮点误差建模。假设有一个任意函数 $y = f(x)$，其中 x 以 IEEE 754 单精度格式表示。假设 x中存在浮点误差 $\sim h$，我们定义 $f(x) = f(x+h)$。符号泰勒级数展开得到：

$$f(x+h) = f(x) + \frac{h}{1!}f'(x) + \frac{h^2}{2!}f''(x) + \frac{h^3}{3!}f'''(x) + \dots$$

这里，$f'(x)$ 表示函数相对于 x 的导数。将级数近似到一阶后，得到：

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

f 中浮点绝对误差的表达式为：

$$A_f = |\widetilde{f}(x) - f(x)| \Rightarrow A_f = |\frac{h}{1!}f'(x)| \Rightarrow A_f = |hf'(x)|$$

为确定 f 中浮点绝对误差的最大值，我们写作 $h = \epsilon_m|x|$，其中 ϵ_m 称为 机器精度。机器精度给出了浮点变量因舍入而产生的最大相对表示误差。它取决于具体机器，并且在大多数编译器中遵循 IEEE 标准。因此，绝对误差可写为：

$$A_f = |\epsilon_m|x|f'(x)| \tag{1}$$

对于大多数较小规模的情况，这种误差模型已经足够，并且可以为程序中允许的最大浮点误差给出宽松的上界。在这里，自动微分技术可以高效计算 $f'(x)$，并扩展到实际工作流。

B. 自动微分基础

自动微分以具有有意义的可微分属性的程序代码作为输入，并生成附加了前推算子或拉回算子 [10] 的新代码。用于生成前推算子、以捕捉从输入到输出的敏感度的自动微分模式，通常称为前向模式。捕捉从输出到输入的敏感度的自动微分模式是

称为 反向 模式、伴随 模式或反向传播。拉回的计算优势在于，它提供了一种非常高效的方法来计算函数的梯度，其相对时间复杂度与输入大小无关 [11]。

AD 工具可以根据程序执行前所完成的工作量进行分类。AD 通常首先构建计算图，即对输入应用数学运算所形成的有向无环图。在一个极端情况下，追踪或磁带记录方法会在每次调用函数时，于执行期间构建并处理计算图。相比之下，源代码变换方法会在编译时尽可能多地完成工作，从而只生成一次导数。追踪方法更易于实现，也更容易应用到现有代码库中。源代码变换方法的性能更好，但通常只覆盖语言的一个子集。**追踪：** 在运行时将计算操作的线性序列记录到磁带中。控制流会被展平，以生成导数。典型的实现方式是运算符重载，即定义一种为基本运算重载了运算符的特殊浮点类型。算法通过调用特殊函数来使用这种类型，从而触发微分。基于追踪的 C++AD 工具很多，其中包括 ADOL- C [12],和 Adept [13]。由于导数是在运行时生成的，即时微分过程受到限制，必须快速执行优化。元编程技术（例如表达式模板）可以缓解这一问题，但它们无法跨语句进行优化，通常也无法处理控制流 [13], [14]。**源代码变换：** 在编译时构建计算图并生成导数。可以应用更多编译时优化，例如重新组织或计算简单的常量表达式，以及进行公共子表达式消除。源代码变换更难实现，因为它需要投入大量工作来开发和维护语言解析器。Tapenade [15] 是源代码变换工具的一个示例，它为 C 和 Fortran 配备了自定义解析器。源代码变换工具通常不支持完整的语言功能集（例如，某些语言惯用法尤其难以进行微分）。

过去，工具开发者往往需要在易用性、性能和集成便捷性之间做出取舍。如今，随着语言支持的改进，自动微分得以避免这类取舍。近期，Clang 等生产级编译器允许工具复用语言解析基础设施。Enzyme [16] 和 Clad [8] 就是利用这类现有解析器的基于编译器的自动微分工具。

III. 使用 CHEF-FP 进行基于自动微分的浮点误差估计

处理精度要求较高的浮点应用程序时，一个重要环节是找出敏感区域（即更容易产生误差的区域），以制定合适的缓解策略。基于自动微分的敏感性分析在这方面非常有用。它能够找出应用程序中对整体浮点误差贡献较大的特定变量或代码区域。

变量的敏感度 x 对浮点误差（S_x）可根据式中描述的默认误差模型推导得出：1 ：

$$S_x = |xf'(x)|$$

变量 x 对函数（无论是例程还是整个程序）中浮点误差的总贡献会随着 S_x 的增加而增加。研究函数定义域内这些敏感度值的变化趋势，可以深入了解函数的数值稳定性，并有助于确定不稳定性的可能原因。敏感度值还可作为一类基于类型的优化的指导依据，这类优化称为混合精度调优。

混合精度调优包括将某些变量降级到较低精度，同时避免对应用程序整体精度造成严重影响。相应地，对于对精度有显著影响的变量，则需要保留或提升其精度。只有当调优前后的精度差异小于指定阈值时，混合精度调优配置才有效。维持这一要求的一种有效方法是分析所有输入变量和中间变量的敏感度，并选取敏感度较低的变量进行降级。将降级变量的浮点误差贡献累加后，与阈值进行比较。当累积误差达到阈值要求时，即可得到混合精度配置。

第 II-A 节中描述的基于泰勒的方法 需要对浮点变量及其各自的伴随量的赋值进行建模。伴随累积模式天然提供了这种映射。CHEF-FP 的实现会从 Clad 导出这些信息。Clad 既可以作为编译降级流水线的一部分使用，也可以生成源代码，再由另一种编译器工具链编译。Clad 实现了前向模式和伴随模式自动微分，并提供灵活的扩展系统，使用户代码可以订阅伴随量创建过程中的事件。

基于编译器的自动微分工具可以在不同的程序表示层级上运行，每种实现都有各自的优缺点。例如，Clad 在 Clang 的高级表示上实现自动微分，从而利用更好的诊断功能、支持编译时编程、生成易于理解的源代码，并使用标准优化流水线。这些特性对于基于自动微分的浮点分析至关重要。如果自动微分在优化流水线之前运行，不安全的优化可能会引入额外的浮点误差。像 Enzyme 等工具那样在优化之后运行自动微分，可以避免这些误差。在这种情况下，即使是标准优化也可能破坏可微性。能否找到一种结合这两种方法优势的途径，仍有待探索。

尽管存在许多基于自动微分的误差分析方法，但它们通常需要大量人工修改代码，或依赖冗长的工具链来实现自动化。在这项工作中，我们提出了 CHEF-FP，这是一种基于自动微分的浮点误差估算框架，所需的人工集成工作更少，并与 Clad 一同打包，从而省去了搭建冗长工具链这一繁琐任务。该工具与编译器紧密集成，而且浮点误差注释被直接构建到代码的导数中，因此能够进行强大的编译器优化；与 ADAPT 等当前最先进的工具相比，这些优化显著缩短了分析时间 [5]。

使用源代码级自动微分工具（如 Clad）作为后端的另一个优势是，它们能够提供重要的源代码信息。例如，高层表示使我们能够识别浮点误差，并将其归因于源代码中可见的变量。对于低级工具而言，恢复这类信息（变量名、ID、源代码位置等）会很困难，因为优化可能会消除感兴趣的变量。Clad 还能够识别特殊结构（如循环、Lambda 表达式、函数、if 语句等），并据此调整误差代码生成。尽管低级工具也能获取这些信息，但在这一层级上提取和处理它们要困难得多。

A. 使用自动微分的源代码变换浮点误差估计

算法 1描述了一种变换，将源代码变换自动微分引擎中可用的变量-伴随映射与浮点误差估计框架连接起来。对于每个标注为误差估计的函数，我们在进行微分（*AdjointAD*）时使用该映射（*NewFunction*），将控制权交给误差模型（*AssignError*），以插入额外指令。最后，我们

算法 1 误差估计生成过程

要求： 由 $estimate\ \ error$ 函数选择
确保： 生成误差估计函数“$function$”

对于所有 $function$ 在 误差估计.函数中**执行**
  $map\langle variable, adjoint\rangle \leftarrow$ ADJOINTAD($function$)
  $function \leftarrow$ NewFunction($function,\ \ map$)
  对所有 $variable,\ \ adjoint$ 中 map **执行**
    ASSIGNERROR($function,\ \ variable,\ \ adjoint$)
  循环结束
  FINALIZEEE($function$)
循环结束

计算总误差（*FinalizeEE*），可以让自定义模型将其作为输出参数传递，或打印在屏幕上。该算法的一个重要特点是，不会对 FPassignments 的右端结构施加限制。对于某些函数的实现，这意味着右端可以包含任意长的表达式。因此，较长表达式的误差贡献计算精度较低。此特性为用户提供了隐式控制能力，使其可以手动缩短表达式。生成代码取决于 EE模型的具体实现；不过，为了更好地理解这一过程，我们可以概括列出一种可能的通用形式。

B. 生成函数的语法结构示例

算法 1 将图 1 中的输入程序转换为另一个如图 2所示的程序。图 1表示一个以计算机编程语言编写的函数，该函数接受 P 参数并返回 T类型的结果。在不失一般性的情况下，其主体由以下指令组成：L_i、$\forall i \in [1..n]$，其中 $S = f_i(S)$ 操作内部状态 S并在 $result(S)$中返回最终状态 S。此表示法不排除可以用足够长的 L_i线性序列表示的控制流结构。

自动微分的伴随累积模式从函数的（$FuncName$）输出开始，朝函数输入（P）方向计算偏导数。这需要以逆序执行函数。图2 展示了自动微分的伴随累积变换，即可能经过扩展的指令序列 L_i。扩展指令记录了一个

函数 $FuncName(P): T$	
	$Initialize\ S\ with\ P$
L_1:	$S = f_1(S)$
	...
L_i:	$S = f_i(S)$
	...
L_n:	$S = f_n(S)$
	返回 $result(S)$

图 1：原始函数 $FuncName$ 的结构

内部状态的子集（表示为 $out(L_i) \subset S$，其取决于 f_i），这是在.....时保留语义所必需的

函数 $FuncName(P, E): void$		
	$Initialize\ S\ and\ \overline{S}$	
L_1:	压入$(out(L_1))$; $S = f_1(S)$	*forward sweep*
	...	
L_i:	推送$(out(L_i))$; $S = f_i(S)$	
	...	
L_{n-1}:	压入$(out(L_{n-1}))$; $S = f_{n-1}(S)$	
$\overleftarrow{L_n}$:	$\overline{S} = \overline{S} \times f_n'(S)$; AssignError($\overleftarrow{L_n}$)	*backward sweep*
	弹出$(out(L_{n-1}))$	
	...	
	弹出$(out(L_i))$	
$\overleftarrow{L_i}$:	$\overline{S} = \overline{S} \times f_i'(S)$; AssignError($\overleftarrow{L_i}$)	
	...	
	弹出$(out(L_1))$	
$\overleftarrow{L_1}$:	$\overline{S} = \overline{S} \times f_1'(S)$; AssignError($\overleftarrow{L_1}$)	
	$E =$ FINALIZEEE($\overline{FuncName}, \overline{S}$)	

图 2：误差估算函数的结构$FuncName$

指令以逆序执行。一种常见的实现机制是使用 LIFO 结构，例如栈，并通过压入 或弹出操作插入压入 或弹出 操作。这种变换称为正向扫描。反向扫描 负责为每条指令 $\overleftarrow{L_i}$计算其伴随量（S）。S 可能需要使用弹出 操作来恢复已更改的状态 S，以便正确计算 $f_i'(S)$。

我们的伴随累积模式扩展增加了 3 个元素：另一个输出参数 $E$to $FuncName$，用于建模总

误差；对每次赋值调用回调 *AssignError* ，并将变量和其伴随作为参数；以及一个*FinalizeEE* 函数，用于计算总误差 E。

C. 变换的结构操作语义

图中伪代码到其误差估计形式的变换1（如图2所示）可以用结构操作语义符号来描述。与 Hascoet 和 Pascual 描述自动微分语义的工作类似 [15]，我们可以扩展这些规则来描述误差估计专用的变换。受篇幅限制，我们仅展示支持浮点误差估计的基本规则。这些规则描述了如何扩展通用的源代码变换自动微分框架，以支持基于自动微分的浮点分析。与自动微分相关的其余规则已见 [15]。

$$\text{S1:}\ \frac{\begin{array}{c} \mathit{isEstErrFunction}(FuncName) \\ FuncName \xrightarrow{newFunction} \overline{FuncName} \\ Params \to \overline{Params} \quad Locals, Params \to \overline{Locals} \\ Stmts \to \begin{bmatrix} \overrightarrow{Stmts} \\ \overleftarrow{Stmts} \end{bmatrix} \end{array}}{\begin{array}{l} \texttt{function}\ FuncName(Params) : T\ \{ \\ \quad Locals;\ Stmts \\ \} \\ \to \texttt{function}\ \overline{FuncName}(\overline{Params}, E : T) : void\ \{ \\ \quad \overline{Locals};\ \overrightarrow{Stmts};\ \overleftarrow{Stmts}; \\ \quad E = \texttt{FinalizeEE}(FuncName) \\ \} \end{array}}$$

在规则 S1 中，满足以下条件时，前提成立：(i) 当 *FuncName* 被选为误差估计候选项时，谓词*isEstErrFunction* 为真；(ii) *newFunction* 可以创建一个$FuncName$；(iii) *Params* 和 *Locals* 可以进行变换；(iv) 原始函数的语句序列已被变换为一次前向扫描和一次反向扫描。当所有前件均成立时，会在反向扫描末尾添加 *FinalizeEE* 。

$$\text{S2:}\ \frac{\begin{array}{c} \mathit{isLive}(Stmt) \quad \mathit{isDiff}(Ref) \\ Stmt \to Ref = Expr \\ \texttt{typeof}(Ref) \xrightarrow{newLocal} \overline{Var} \\ Expr, \overline{Var} \xrightarrow{Expr} \overline{Stmts} \\ Ref, Expr, \overline{Var} \xrightarrow{buildAssignError} Est \end{array}}{\begin{array}{l} Ref = Expr \\ \quad \to \begin{bmatrix} \texttt{Push}(Ref);\ Ref = Expr;\ \overline{Var} = 0 \\ \texttt{Pop}(Ref);\ \overline{Stmts};\ Est \end{bmatrix} \end{array}}$$

在规则 S2 中，满足以下条件时，前提成立：(i) 当语句*Stmt* 对导数计算有用时，*isLive* 为真；(ii) 当内存位置与导数计算相关时，*isDiff* 为真；(iii) *newLocal* 可以创建一个与*Ref*类型相同的变量；(iv) *Expr* 可以对*Expr*求导；(v) *buildAssignError* 可以生成 EE 指令*Est*。所有前提均满足时，*AssignError* 会扩充反向扫描。*Ref* 表示一个可以存储值的内存位置。

$$\text{S3:}\ \frac{\begin{array}{c} \mathit{isLive}(Stmt) \quad \neg\mathit{isDiff}(Ref) \\ Stmt \to Ref = Expr \\ Ref, Expr \xrightarrow{buildAssignError} Est \end{array}}{Ref = Expr \to \begin{bmatrix} \texttt{Push}(Ref);\ Ref = Expr \\ \texttt{Pop}(Ref);\ Est \end{bmatrix}}$$

$$\text{S4:}\ \frac{\begin{array}{c} \neg\mathit{isLive}(Stmt) \quad \mathit{isDiff}(Ref) \\ Stmt \to Ref = Expr \\ \texttt{typeof}(Ref) \xrightarrow{newLocal} \overline{Var} \\ Expr, \overline{Var} \xrightarrow{Expr} \overline{Stmts} \\ Ref, Expr, \overline{Var} \xrightarrow{buildAssignError} Est \end{array}}{Ref = Expr \to \begin{bmatrix} \overline{Var} = 0 \\ \overline{Stmts};\ Est \end{bmatrix}}$$

接下来的两条规则 S3 和 S4考虑 *isLive*和*isDiff* 谓词值的变化。

当*isDiff* 或*isLive* 为 false 时，我们仍会创建伴随量，并插入*AssignError* 来捕获变量的误差贡献。函数调用和参数传递也可以通过类比来表达。

D. 编程模型与框架设计

Clad 的回调系统允许创建可扩展生成代码的扩展。我们利用这一能力构建了一个轻量级框架，以便在 Clad 生成的伴随代码中插入浮点误差估计代码。CHEF-FP 利用 Clad 灵活的设计，并将自身作为原生扩展添加，在微分过程中合成误差估计代码。图3 概述了其高层设计。CHEF-FP 的实现大致分为误差估计模块和误差模型。

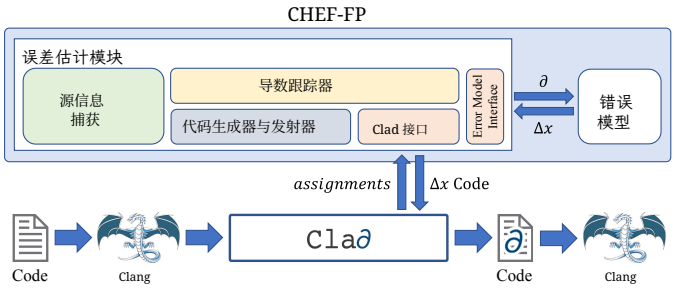


图3：**CHEF-FP 中的浮点误差估计生成工作流**。Clang 的编译流水线控制权被 Clad截获，检测到对 estimate_error() 的调用后，会实例化 误差估计模块并设置回调。随后，误差估计模块 监听来自 Clad 伴随模式的回调，并负责使用误差模型中定义的误差估计代码增强导数主体，然后将控制权传回 Clad。误差估计模块 还负责缓存值、跟踪导数、捕获所需的结构信息和源代码信息，以及累积误差。

误差估计模块：误差估计模块是构成CHEF-FP 的主要组件。通过注册对 estimate_error() 的调用（用户借此标注感兴趣的函数），实例化了 误差估计模块。该模块设置误差估计回调，并将控制权交还给 Clad。误差估计模块监听来自Clad伴随模式的回调，并在触发一个有意义的（从误差估计角度看）回调时接管控制权。误差估计模块通过添加从误差模型接收的误差估计表达式来扩展导数函数体，并将控制权交还给 Clad，直到下一次调用回调。因此，它负责与 Clad 对接，定义各种方法以促进浮点误差估计代码的生成与输出，并与 误差模型 交换信息，从而生成经过特定扩展的代码。误差估计模块还会缓存值、跟踪导数、捕获所需的结构信息和源信息，并累积误差，以提供最终的浮点误差估计。

误差模型： 误差模型定义了一个接口，用于描述要生成的误差表达式。生成导数时，它会接收导数表达式，并返回相应的误差表达式。默认情况下，系统会合成对用户定义函数 `getErrorVal` 的调用，让用户能够使用纯 C/C++ 代码编写公式。此外，还提供了一个可编程的附加接口，用于生成更高级的调用，以导出更多伴随信息，从而纳入其他高级自定义误差模型。此接口可完全自定义，使浮点误差分析更贴合具体应用。清单 1 通过一个最简示例展示了该编程模型。

清单 1 CHEF-FP 用法的最简示例。

```
浮点 func(浮点 x,浮点 y) {
  浮点 z;
  z = x + y;
  返回 z;
}
int main() {
  // 调用目标函数的误差估计。
  自动微分 = Clad::估计_error(func);
  // 声明输入及其导数
  // 输出和最终误差输出。
  浮点数x = 1.95e-5, y = 1.37e-7;
  浮点dx = 0, dy = 0;
  双精度浮点_误差 = 0;
  // 执行生成代码。
  df.execute(x, y, &dx, &dy, 浮点_error);

  // fp_error 现在包含 func 的误差。
  std::cout << "函数中发生错误： " << 浮点_误差;
}
```

E. 实现

CHEF-FP 在 Clad 中注册了一个应用程序接口（估计_误差），可使用默认误差估计模型计算给定函数中的浮点误差。清单 1展示了 CHEF-FP 的典型调用示例。

对于更复杂的分析，可能需要更改底层误差模型，以获得令人满意的估算结果。只需实现`FPErrorEstimationModel` 接口，并采用适当的误差模型，将其编译为共享库，再将该库传递给 CHEF-FP，即可实现这一点。清单2中介绍了此类实现的一个示例。这里，`AssignError` 会构建代码

清单 2 用于声明自定义模型类的模板。

```
结构体 自定义模型： 公共
↪  FPErrorEstimationModel {
  CustomModel(DerivativeBuilder& builder)
      : FPErrorEstimationModel(builder) {}

  // 返回误差表达式以供
  // 为每个变量赋值计算。
  Clang::Expr* AssignError(StmtDiffrefExpr,
↪  const 字符* 名称)覆盖;
};
```

表达式（以 Clang 的内部表达式类型表示），之后误差估计模块可以将其生成到代码中。此函数公开了三个值，可用于实现自定义模型——变量的名称、变量本身及其导数。

清单 3 构建外部函数调用的 AssignError 示例实现。该实现会生成对用户定义函数`getErrorVal`的调用。

```
namespace clad{ double getErrorVal(double dx, double
x, const char* name) { ↪return dx * (x - (float)x); } }
clang::Expr* CustomModel::AssignError(StmtDiff
refExpr, std::string name) { ↪// 构建一个类似向量的容器,
用于存储 // 函数调用的参数。 llvm::SmallVector<
clang::Expr*, 3> params{ refExpr.getExpr_dx(),
refExpr.getExpr(), utils::CreateStringLiteral( m_Context,
↪name)};↪// 返回对 getErrorVal 的调用。 return
GetFunctionCall("getErrorVal", "clad", params); ↪}
```

实现 AssignError 大致有两种方式：构建算术表达式，或构建外部函数调用。这两种方法都需要构建可赋值表达式。前一种方式需要直接构建要赋值的算术表达式，因此适用范围相当有限。此外，每次修改 AssignError 时都必须重新构建模型。虽然我们提供了一个便于构建简单表达式的应用程序接口，但仅凭单个表达式来实现复杂模型并不直观。因此，只要函数的返回类型与要赋予误差的变量兼容，我们就可以将对外部函数的调用构建为 validerror 模型。这种方式允许用户将误差模型定义为常规的 C++ 函数，从而实现计算复杂度更高的模型。

清单 3 展示了如何实现任意函数 `getErrorVal` 并将其用作自定义误差模型。在该清单中，我们构建了 ADAPT-FP 中使用的模型，其数学表示如下：

$$\Delta = \sum_{i=1}^{n} \frac{\delta f}{\delta x_i} * (x_i - (float)x_i), \tag{2}$$

其中，Δ 是函数 f 中因所有 x_i 的舍入误差而累积的误差。这里，分配给每个变量的误差是其单精度值与双精度值之差。请注意，此模型要求所应用的所有函数均采用双精度或更高精度。

还可以通过修改`getErrorVal`函数来更改示例外部误差函数（即 3 中定义的函数）的签名和组成，方法是修改 `AssignError` 函数。这使用户能够添加由 Clang 暴露的变量的相关元信息。例如，用户可以实现`AssignError`，以构造对一个可接收变量源代码位置的误差函数的调用。此信息可用于识别代码中更容易产生误差的特定区域。

IV. 实验

我们使用五种不同的算法，将 CHEF-FP 与当前最先进的 ADAPT 进行比较：弧长、辛普森法、*k* 均值聚类、*HPCCG*和布莱克-斯科尔斯。对于前 4 个基准测试，我们使用方程2中描述的误差模型。运行时间通过*Google* 基准测试测量，峰值内存则通过*GNU time*测量。基准测试在普林斯顿*Tiger* 集群 [17]上进行，该集群配备一颗 2.4GHz Intel Xeon Gold 6148 CPU 和 188 GB RAM。使用劳伦斯利弗莫尔国家实验室的 Quartz [18] 系统也获得了类似的基准测试结果。该系统是一个集群，由 Intel Xeon E5-2695 处理器组成，核心频率为 2.1 GHz，每个节点配备 128 GB 内存。

基准测试	阈值	实际误差	估算值错误	加速比
弧长	1e-05	3.24e-06	3.24e-06	1.11
辛普森法	1e-06	7.80e-08	1.32e-07	2.25
k-均值	1e-06	0.00e+00	0.00e+00	-
HPCCG	1e-10	5.21e-12	5.92e-11	1.08

表 I：**基准测试混合精度版本的误差和性能测量**。该表展示了原始程序的混合精度版本中实际误差与 CHEF-FP 估算误差的对比。它还展示了混合精度变体的执行加速比。对于 *k-Means*，CHEF-FP 针对所定义阈值确定的混合精度配置没有带来加速比。

对于以下所有基准测试，我们展示了分析时间和内存占用的减少，同时仍能生成与 ADAPT 等工具相当的结果。我们还利用本节探讨如何扩展某些示例的分析，以及如何使用 CHEF-FP 执行更复杂的近似分析。最后，我们在表 I 中展示 CHEF-FP 如何在几乎无需用户干预的情况下，生成正确的浮点分析结果。

1）弧长：弧长函数通过对曲线上的多个点进行采样，并将曲线上相邻两点之间的直线距离相加，来近似计算曲线的长度（L）。

$$L = \lim_{n\to\infty} \sum_{i=1}^{n} \sqrt{\Delta x^2 + \Delta {y_i}^2}$$

我们改变 n，即弧长算法的运行迭代次数，以对 CHEF-FP 和 ADAPT 进行基准测试。图 4 比较了对算法进行混合精度分析所需的时间和内存使用量。ADAPT 在 10^8 次迭代时没有数据点，是因为它耗尽了内存。CHEF-FP 的内存占用更低，因为误差计算被内联到了梯度函数中。

2) 辛普森法：辛普森法是一种迭代算法，通过对给定区间内多个小区间上的积分求和，来近似函数的积分：

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\left[f(a) + f(b) + 4\sum_{i=1,3,5}^{2n-1} f_i + 2\sum_{i=2,4,6}^{2n-2} f_i\right]$$

这里，$f_i = f(a + ih)$，$h = (a + b)/2n$，以及 n 表示迭代次数。图5 根据时间和内存使用量对 ADAPT 与 CHEF-FP 进行了比较。与弧长算法类似，我们改变迭代次数来对这两种工具进行基准测试。与使用更高精度的同一程序相比，CHEF-FP 推荐的混合精度配置可实现 2.25 倍的加速比。它还能够预测应用程序混合精度版本的实际误差，如表I所示。

3) *k*-均值聚类：Rodinia 基准测试套件的一部分 [19]，该 *k*-均值聚类算法用于将多个数据点分组为 k 多个簇。我们对欧几里得距离函数进行了插桩，因为它是该应用程序的主要计算热点。与之前的基准测试类似，我们在图6中比较了 CHEF-FP 与 ADAPT 的性能。

欧几里得距离函数有三个主要变量：属性、簇和总和。它可以表示如下：

$$sum = \sqrt{\sum_{i=0}^{n} (attributes_i - clusters_i)^2}$$

Clad 对 属性 估算的误差为 0，因为基准测试的输入数据保留了四位小数。对 簇 和 总和估算的误差高于表中为 *k*均值设定的阈值I。因此，CHEF-FP 仅建议将 属性 转换为较低精度。为了验证 CHEF-FP 的误差估算，我们进一步逐一将各个变量转换为较低精度，并测得由此引入的实际误差。这些配置在 10^6 个数据点上执行，结果如表III所示。

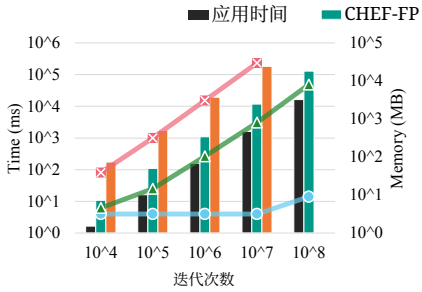


图 4：弧长

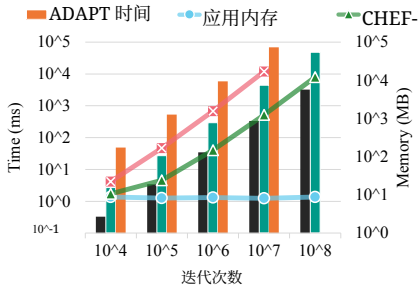


图 5：辛普森法

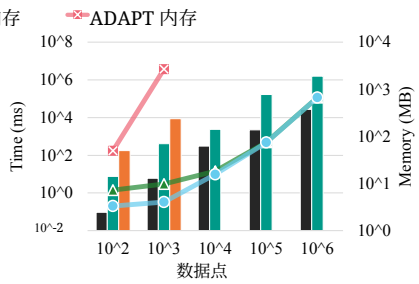


图 6： k-均值算法

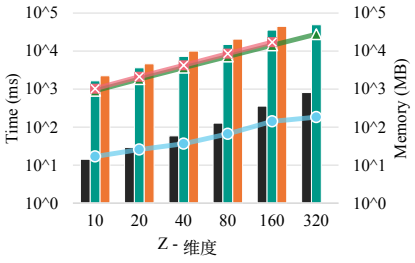


图 7：HPCCG

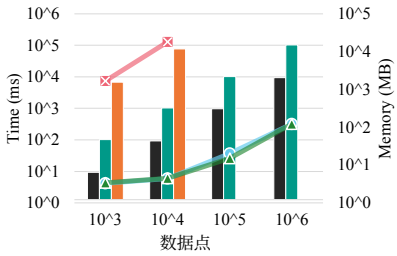


图 8：布莱克-斯科尔斯

基准测试	时间	内存
弧长	1.61 倍	1.95x
辛普森法	2.17 倍	1.44 倍
k-均值	2.02x	4.44 倍
HPCCG	1.03x	1.02x
布莱克-斯科尔斯	1.76x	6.32x

表 II：性能提升

图 4 至 8 展示了对 CHEF-FP、ADAPT 和原始函数进行基准测试的结果。 函数下方的标签显示了正在进行基准测试的算法。折线表示峰值内存使用量，柱状图表示对给定算法进行浮点误差分析所花费的时间。 **表 II 总结了 CHEF-FP 相较于 ADAPT 的性能提升。** 这些提升以浮点误差分析的“提升倍数”表示，代表所有数据点上的平均提升。基准测试结果表明，CHEF-FP 的性能优于 ADAPT。ADAPT 是当前最先进的基于自动微分的浮点误差估计工具，而 CHEF-FP 得出的混合精度分析结果与 ADAPT 的分析结果一致。

低精度变量	实际误差	估计误差
属性	00e+00	00e+00
簇	3.67e-04	9.35e-04
sum	8.33e-04	7.08e-03
全部 3 个	2.40e-03	8.01e-03

表 III：**k 均值 – 各种混合精度配置的误差测量。**我们将 3 个变量（属性、聚类 和 总和）逐一降至较低精度，并将所得误差与 CHEF-FP 的估计值进行比较。

4）HPCCG：作为 Mantevo 基准测试套件的一部分，HPCGG 是一段用于三维烟囱域的简单共轭梯度基准测试代码，经过转换后为单线程代码。分析过程中，输入从基础维度 $20 \times 30 \times 10$ 扩展到建议的大小 $20 \times 30 \times 160$，然后进一步扩展到 $20 \times 30 \times 320$。图 7 中的结果显示，ADAPT 在 $20 \times 30 \times 320$ 维度下耗尽了内存。

CHEF-FP 还用于分析 HPCCG，探索通过循环穿孔进行优化的可能性。我们已经生成了中间变量的误差，因此只需稍作调整，即可输出每次迭代中各变量的敏感度。我们分析了应用程序整个运行过程中各变量敏感度的变化，发现经过约 60 次迭代后，所有变量的敏感度都低于设定的阈值。变量的归一化敏感度

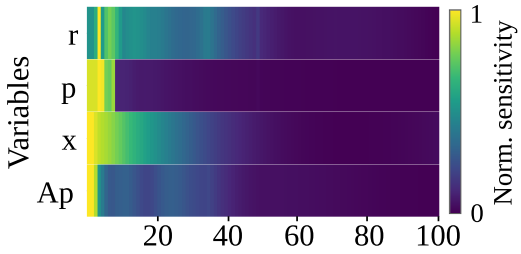


图 9：**HPCCG 变量热图。** 这展示了变量 r、p、x和 Ap 在每次迭代中的归一化敏感度。该敏感度特征可用于确定更精细的优化。

以热力图的形式显示，如图9所示。根据这些发现，我们将 HPCCG 的主循环分为两段：第一段以高精度完整运行前 60 次迭代，第二段则以较低精度运行剩余迭代次数。此配置可带来8% 的加速比，如表I所示。

5) 布莱克-斯科尔斯： Parsec 基准测试套件的一部分 [20],布莱克-斯科尔斯方程是一种微分方程，用于描述期权价值随标的资产价格变化而发生的变化。为与 ADAPT 进行比较，我们分析了根据影响股票的因素计算期权价格的函数。ADAPT 无法

扩展到 10^4 个数据点，涵盖我们的基准测试。结果如图8所示。对于此基准测试，CHEF-FP 和 ADAPT 均显示，大多数中间变量对误差非常敏感，且无法确定有效的混合精度配置。

此外，我们利用 CHEF-FP 的可定制性，分析了使用 Paul Minero 的 FastApprox 库 [21] 替代标准 C 数学库所带来的性能提升。FastApprox 库提供了各种数学函数的近似版本，以精度换取性能。CHEF-FP 可以确定在任何程序中用近似版本替换标准数学函数版本所引入的误差。我们在 Black-Scholes 应用程序中找到了三个在 FastApprox 库中有近似版本的数学函数。我们确定了这些函数的输入，并建立了一张映射表，将它们的名称映射到其输入的函数。自定义模型使用该映射表，将每个变量与相应的函数及其近似版本相匹配。随后，自定义模型利用这些值准确估算应用程序中由近似处理导致的误差。自定义模型的算法如算法2所示。FastApprox 在 Black-Scholes 应用程序中的估算误差、实际误差和加速比如表 IV所示。

算法 2 估算基于近似的误差。

要求： 输入变量 x 及其名称 $name$，偏导数 x 相对于函数 dx的导数，以及一个感兴趣变量映射 $S : name \rightarrow$ 函数名称

```
Δ ← 0
如果 name 包含在 S 则
    fName ← S.GETVALUE(name)
    Δ ← EVAL(fName, x)–EVALAPPROX(fName, x)
结束 if
xApproxError ← |dx ∗ Δ|
REGISTERERROR(name, xApproxError)
返回 xApproxError
```

V. 讨论

与 ADAPT 相比，CHEF-FP 始终表现更佳，同时为混合精度调优提供了类似的建议，并带来相近的性能收益。CHEF-FP 提供了一种高效、直接且灵活的方式，用于分析复杂 C++ 应用程序中的浮点误差。我们展示了如何使用 CHEF-FP 对 4 个基准测试——弧长、辛普森法、*k*-均值 和 *HPCCG*进行混合精度调优和敏感性分析，并将其用于 1 个基准测试——布莱克-斯科尔斯的近似分析。本节将更详细地讨论实验结果和当前的局限性。

A. CHEF-FP 的结果摘要

CHEF-FP 为给定阈值下的弧长和辛普森法基准测试确定了稳定的混合精度配置；转换为这些混合精度配置后，两个应用程序都获得了明显的加速比，结果汇总于表I中。对于 *k*-均值基准测试，CHEF-FP 未能确定能够实现加速比的混合精度配置（阈值为10^{-6}）。不过，CHEF-FP 为应用程序在不同混合精度配置下的误差提供了良好的上界估计（见表III）。

对于 *HPCCG* 基准测试，该工具用于分析主循环中各变量的敏感度。这使我们得以发现一种混合精度配置：将主循环拆分为两部分，前 60 次迭代采用更高精度，其余部分采用较低精度。最后，我们还展示了 CHEF-FP 的灵活性：利用其自定义模型支持，对布莱克-斯科尔斯 基准测试进行了近似误差分析。CHEF-FP 能够准确量化一组特定函数中与近似有关的误差。此外，CHEF-FP 生成了两种不同的近似配置，并在表 IV中展示了相应的估计误差报告。

B. 当前局限性

量化类型转换的开销：混合精度配置的性能可能比高精度版本更差。通常，这是由于降低某些变量的精度后，代码中引入的隐式类型转换所带来的开销。应对这些开销的一种方法是通过计数器对其进行跟踪；例如，可以使用 Clang 的 AST 匹配器实现一个简单的隐式类型转换计数器。

分析数据集的多样性：此处形成的混合精度配置的结果取决于输入。要形成通用的混合精度配置，重要的是在具有代表性的一组输入上分析该应用程序。

混合精度配置的源代码重写：CHEF-FP 提供敏感度概况和误差估计，以指导混合精度重新实现过程。目前，我们手动重写源代码，以实现 CHEF-FP 建议的混合精度配置。我们可以使用源代码变换工具，例如 Typeforge [22], 来自动生成混合精度代码。作为未来的工作，可以通过结合决策与代码生成，并利用运行时的误差信息，即时优化敏感度较低的代码区域，使其以较低精度运行，从而实现这一过程的自动化。

编译器优化： 某些不安全的浮点优化（例如 *-ffast-math* 或 *-fp-model fast*）可能导致 CHEF-FP 预测的误差与实际误差不同。目前，CHEF-FP 无法区分优化引入的误差与预期的浮点误差。由于这些优化发生在导数生成之后，某些替换优化甚至可能导致导数本身错误，进而使底层的误差传播也不正确。CHEF-FP 的用户应谨慎对待这些优化，因为它们可能导致分析结果不正确。另一种可能影响分析结果的方式，是更改混合精度表达式的中间舍入模式（通过 *-fp-model* 等标志）。更改此设置可能导致

应用程序配置	实际误差			估计误差			加速比
	avg.	*max.*	*acc.*	*avg.*	*max.*	*acc.*	
不含 Fast exp 的 FastApprox	1.16e-04	9.62e-05	1.16e+01	9.16e-04	6.25e-04	9.60e+00	1.14
Fastapprox 搭配快速 exp	5.8e-04	6.9e-04	5.88e+01	3.5e-03	5.0e-03	1.07e+02	1.65

表 IV：**布莱克-斯科尔斯——基于 FastApprox 的各种配置的误差与性能分析**。此表展示了近似版本应用程序的实际误差以及 CHEF-FP 估算的误差。具体而言，它概述了 1000 个数据点上的平均误差、最大误差和累计误差。第一行中，原始程序使用了近似版本的 *log* 和 *sqrt* 函数。第二行中，应用程序使用了近似版本的*exp*。

程序的混合精度版本性能下降。我们建议使用*source* 模式对中间计算进行舍入，以获得一致的结果。

VI. 相关工作

文献中提出了许多浮点误差分析工具，包括静态和动态技术。动态方法需要运行程序以收集分析所需的信息。Brown 等人 [23] 设计了基于 Valgrind 构建的 FloatWatch，用于确定能否通过采用较低精度表示或定点算术来优化浮点运算。该工具通过同时执行单精度和双精度计算，跟踪两者之间的最大差值。Benz 等人 [24] 提出了一种方法，将每项浮点计算并行地以更高精度执行，以帮助程序员定位浮点精度问题。Lam 等人 [25] 提出了一种使用二进制分析工具 DynInst 的动态方法，用于检测浮点抵消。An 等人 [26]开发了基于 DynInst 的动态二进制分析工具 FPInst，通过应用简单的误差累积公式并在整个程序中跟踪误差来计算误差。静态分析工具，如 FPTaylor [4], SATIRE [27], 和 Precisa [28], 使用全局优化器来估算舍入误差的上界。Gappa [29] 利用区间算术自动完成误差评估与传播。静态分析方法能够提供严格的误差分析，但目前仅应用于小型基准测试。SEESAW [3] 采用符号伴随模式自动微分，为输入值区间给出更紧的误差界，并已证明可用于实际的高性能计算基准测试。这些方法均未针对混合精度。

已有多项工作评估程序能否利用混合精度。大多数技术采用基于搜索的优化，选择合适的混合精度程序版本，使其满足用户指定的误差阈值。Lam 等人 [28], [30] 提出了 CRAFT，它利用搜索算法自动识别可采用较低精度的代码区域。Gonzales 等人 [1] 使用差分调试缩小混合精度配置的搜索空间。此方法后来扩展为考虑变量组，从而进一步缩小搜索空间 [31]。Laguna 等人 [32]提出了 GPUMixer，这是一种用于调整 GPU 程序中浮点精度的工具，重点在于性能提升。它通过影子计算分析，计算每个内核因混合精度引入的误差，并使用基于搜索的技术来确定最佳混合精度配置。这些方法通过找出一组可采用单精度的变量，同时让其余变量保持双精度来实现。基于搜索的技术有一个缺点：它们需要多次运行应用程序，而且探索搜索空间极其耗时。

过去已有多项工作致力于利用自动微分（AD）分析应用程序，以引入混合精度，并估计低精度表示所导致的误差。自20世纪90年代初以来，AD一直被用于估计数值算法中的舍入误差 [33], , 其中利用AD得到的偏导数来获得由基本舍入误差引起的全局舍入误差的一阶近似。研究人员还结合中值定理开展区间分析，以得到更紧的舍入误差估计上界。随后，Langlois [34] 提出了 CENA 方法，在该方法中引入了一个修正项，用于表示舍入误差对数值算法输出的一阶影响，从而提高估计的准确性。之后，ADAPT [5]利用AD估计误差，通过识别可以应用较低精度且仍处于误差阈值以内的区域，实现了混合精度调优。虽然 ADAPT 为混合精度实现提供了指导，但其中涉及手动注释和代码转换。为解决这一问题，FloatSmith [7] 应运而生；它将 ADAPT 与 CoDiPack（ADtool） [35]以及 Typeforge（基于 Rose [36]编译器）相集成，从而自动化分析数值代码的过程。然而，这条冗长的工具链使相关工作变得缓慢且繁琐。

VII. 结论

本文定义了一套形式化方法，用于扩展自动微分以执行浮点误差分析，并展示了一种高效工具：它采用基于编译器的源代码变换，不会让本已过于复杂的反向累积自动微分模式不堪重负。我们介绍了 CHEF-FP，这是一种灵活、可扩展且易于使用的、基于源代码变换和自动微分的工具，用于分析高性能计算应用中的近似误差。CHEF-FP 在源代码级将误差估计代码注入生成的伴随代码，从而借助编译器优化加快分析速度。与其他浮点误差估计工具相比，这种方案使 CHEF-FP 能够处理更大的内存负载。它通过使用自定义误差模型，灵活控制生成何种估计代码，从而便于探索近似误差分析的其他领域。

我们通过五个基准测试证明，CHEF-FP 能够以高效利用时间和内存的方式执行与 ADAPT-FP 相同的分析。在分析时间方面，对于 Simpsons 基准测试，CHEF-FP 相较于 ADAPT-FP 的最高加速比为 2.17 倍；对于 Black-Scholes 基准测试，内存占用减少了 6.32 倍。我们展示了如何使用 CHEF-FP 执行敏感性分析，并进一步提供了在各种应用程序中进行混合精度调优的建议。我们说明了如何使用该工具准确评估不同的精度配置。最后，我们探索了基于近似的误差估计，并在 *Black-Scholes* 基准测试上评估了由此得到的近似配置，结果显示加速比最高可达 65%。

这项工作的开源成果可通过数字对象标识符获取：10.5281/zenodo.7660443。

VIII. 致谢

作者谨向匿名审稿人致以感谢，感谢他们提出的宝贵意见和建议。本项目获得美国国家科学基金会 Grant OAC- 1931408 和 Cooperative Agreement OAC-1836650 的支持。本材料基于美国能源部科学办公室高级科学计算研究办公室以及劳伦斯利弗莫尔国家实验室-LDRD 计划项目编号 20-ERD-043 的支持开展的工作。本工作由美国能源部主持，在合同 DE-AC52-07NA27344（LLNL-CONF-841475）下，由劳伦斯利弗莫尔国家实验室完成。

参考文献

[1] C. Rubio-Gonz´alez、C. Nguyen、H. D. Nguyen、J. Demmel、W. Kahan、K. Sen、D. H. Bailey、C. Iancu 和 D. Hough，《Precimonious：浮点精度调优助手》，载于 SC'13：高性能计算、网络、存储与分析国际会议论文集。IEEE，2013年，第1–12页。[2] M. O. Lam 和 J. K. Hollingsworth，《细粒度浮点精度分析》，《国际高性能计算应用期刊》，第32卷，第2期，第231–245页，2018年。[3] A. Das、T. Tirpankar、G. Gopalakrishnan 和 S. Krishnamoorthy，《通过符号自动微分分析无循环浮点程序的鲁棒性》，载于2021年 IEEE 国际集群计算会议（CLUSTER）。IEEE，2021年，第481–491页。[4] A. Solovyev、M. S. Baranowski、I. Briggs、C. Jacobsen、Z. Rakamari´c 和 G. Gopalakrishnan，《利用符号泰勒展开严格估算浮点舍入误差》，《ACM 程序设计语言与系统汇刊》（TOPLAS），第41卷，第1期，第1–39页，2018年。[5] H. Menon、M. O. Lam、D. Osei-Kuffuor、M. Schordan、S. Lloyd、K. Mohror 和 J. Hittinger，《ADAPT：将算法微分应用于浮点精度调优》，载于 SC18：高性能计算、网络、存储与分析国际会议。IEEE，2018年，第614–626页。[6] V. Vassiliadis、J. Riehme、J. Deussen、K. Parasyris、C. D. Antonopoulos、N. Bellas、S. Lalis 和 U. Naumann，《面向近似计算的自动显著性分析》，载于2016年 IEEE/ACM 国际代码生成与优化研讨会。IEEE，2016年，第182–193页。[7] M. O. Lam、T. Vanderbruggen、H. Menon 和 M. Schordan，《源代码级混合精度的工具集成》，载于2019年 IEEE/ACM 第三届高性能计算应用软件正确性国际研讨会（Correctness），2019年，第27–35页。[8] V. Vassilev、M. Vassilev、A. Penev、L. Moneta 和 V. Ilieva，《Clad——使用 Clang 和 LLVM 实现自动微分》，《物理学杂志：会议系列》，第608卷，第1期，第 012055, 2015, [Link]。[9]《IEEE 浮点算术标准》，IEEE 754-2019 标准（修订 IEEE 754-2008），第1–84页，2019年。[10] M. Betancourt，《高阶自动微分的几何理论》，arXiv 预印本 arXiv:1812.11592，2018年。[11] A. Griewank 和 A. Walther，《导数求值：算法微分的原理与技术》。SIAM，2008年。[12] A. Griewank、D. Juedes 和 J. Utke，《算法755：ADOL-C：用于对以 C/C++ 编写的算法进行自动微分的软件包》，《ACM数学软件汇刊》（TOMS），第22卷，第2期，第131–167页，1996年。[13] R. J. Hogan，《在 C++ 中使用表达式模板实现快速反向模式自动微分》，《ACM数学软件汇刊》（TOMS），第40卷，第4期，第1–16页，2014年。[14] M. Sagebaum、T. Albring 和 N. R. Gauger，《算法微分反向模式中用于原始值记录的表达式模板》，《优化方法与软件》，第33卷，第4–6期，第1207–1231页，2018年。[15] L. Hascoet 和 V. Pascual，《Tapenade 自动微分工具：原理、模型与规范》，《ACM数学软件汇刊》（TOMS），第39卷，第3期，第1–43页，2013年。[16] W. S. Moses 和 V. Churavy，《无需为机器学习重写外部代码，自动合成快速梯度》，载于《神经信息处理系统进展第33卷》。Curran Associates, Inc.，2020年。[17]《普林斯顿研究计算中心》，[Link]，访问日期：2023-02-20。[18]《利弗莫尔计算中心：劳伦斯利弗莫尔国家实验室的高性能计算》，[Link]，访问日期：2023-02-20。[19] S. Che、M. Boyer、J. Meng、D. Tarjan、J. W. Sheaffer、S.-H. Lee 和 K. Skadron，《Rodinia：异构计算基准测试套件》，载于2009年 IEEE 国际工作负载特征分析研讨会（IISWC），2009年，第44–54页。[20] C. Bienia，《现代多处理器基准测试》，博士论文，普林斯顿大学，2011年1月。[21] P. Mineiro，《Fastapprox》，[Google档案 Link], 2011。[22] N. T. Pinnow、M. Schordan 和 T. L. Vanderbrugger，《Typeforge》，劳伦斯利弗莫尔国家实验室（LLNL），加利福尼亚州利弗莫尔，技术报告，2019年。[23] A. W. Brown、P. H. Kelly 和 W. Luk，《为可重构实现分析浮点数值范围》，载于第一届 HiPEAC 可重构计算研讨会论文集，2007年，第6–16页。[24] F. Benz、A. Hildebrandt 和 S. Hack，《用于发现浮点精度问题的动态程序分析》，《ACM SIGPLAN 通讯》，第47卷，第6期，第453–462页，2012年。[25] M. O. Lam、J. K. Hollingsworth 和 G. Stewart，《动态浮点消去检测》，《并行计算》，第39卷，第3期，第146–155页，2013年。[26] D. An、R. Blue、M. Lam、S. Piper 和 G. Stoker，《Fpinst：使用 Dyninst 进行浮点误差分析》，2008年。[27] A. Das、I. Briggs、G. Gopalakrishnan、S. Krishnamoorthy 和 P. Panchekha，《可扩展且严格的浮点误差分析》，载于 SC20：高性能计算、网络、存储与分析国际会议。IEEE，2020年，第1–14页。[28] M. A. Feli´u、M. Moscato、C. A. Mu˜noz 等，《用于浮点程序舍入误差分析的抽象解释框架》，载于《验证、模型检查与抽象解释国际会议》。施普林格，2018年，第516–537页。[29] M. Daumas 和 G. Melquiond，《含舍入运算符的表达式界限认证》，《ACM数学软件汇刊》（TOMS），第37卷，第1期，第1–20页，2010年。[30] M. O. Lam、J. K. Hollingsworth、B. R. de Supinski 和 M. P. LeGendre，《自动调整程序以实现混合精度浮点计算》，载于第27届国际超算会议国际ACM会议论文集，2013年，第369–378页。[31] H. Guo 和 C. Rubio-Gonz´alez，《利用社区结构调优浮点精度》，载于第27届 ACM SIGSOFT 国际软件测试与分析研讨会论文集，2018年，第333–343页。[32] I. Laguna、P. C. Wood 等，《Gpumixer：面向 GPU 科学应用的性能驱动型浮点调优》，载于高性能计算国际会议。施普林格，2019年，第227–246页。[33] M. Iri，《自动微分与舍入误差估算的历史》，载于 Andreas Griewank 和 George Corliss 编著，第3–16页，1991年。[34] P. Langlois，《CENA 方法的修订版介绍》，博士论文，法国国家信息与自动化研究所，2000年。[35] N. G. M. Sagebaum、T. Albring，《使用 CoDiPack 进行高性能导数计算》，《ACM数学软件汇刊》（TOMS），第45卷，第4期，2019年。[36] D. Quinlan 和 C. Liao，《ROSE 源到源编译器基础设施》，载于与 PACT 联合举办的 Cetus用户与编译器基础设施研讨会，第2011卷。Citeseer，2011年，第1页。