Fast And Automatic Floating Point Error Analysis With CHEF-FP

Garima Singh[⋆§], Baidyanath Kundu[⋆§], Harshitha Menon[†], Alexander Penev[‡], David J. Lange[§], Vassil Vassilev[§⋆]

[⋆]*European Council for Nuclear Research, Espl. des Particules 1, 1211 Meyrin, Switzerland*
[§]*Department of Physics, Princeton University, Princeton, New Jersey 08544, USA*
[†]*Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, California 94551 USA*
[‡]*Faculty of Mathematics and Informatics, University of Plovdiv, 236 Bulgaria Blvd., 4000 Plovdiv, BULGARIA*
E-mail: [⋆]*{garima.singh, baidyanath.kundu, vassil.vassilev}@cern.ch,* [†]*harshitha@llnl.gov,* [‡]*apenev@uni-plovdiv.bg,*
[§]*david.lange@princeton.edu*

2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 979-8-3503-3766-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/IPDPS54959.2023.00105

***Abstract*—As we reach the limit of Moore's Law, researchers are exploring different paradigms to achieve unprecedented performance. Approximate Computing (AC), which relies on the ability of applications to tolerate some error in the results to trade-off accuracy for performance, has shown significant promise. Despite the success of AC in domains such as Machine Learning, its acceptance in High-Performance Computing (HPC) is limited due to its stringent requirement of accuracy. We need tools and techniques to identify regions of the code that are amenable to approximations and their impact on the application output quality so as to guide developers to employ selective approximation. To this end, we propose CHEF-FP, a flexible, scalable, and easy-to-use source-code transformation tool based on Automatic Differentiation (AD) for analysing approximation errors in HPC applications.**

CHEF-FP uses Clad, an efficient AD tool built as a plugin to the Clang compiler and based on the LLVM compiler infrastructure, as a backend and utilizes its AD abilities to evaluate approximation errors in C++ code. CHEF-FP works at the source level by injecting error estimation code into the generated adjoints. This enables the error-estimation code to undergo compiler optimizations resulting in improved analysis time and reduced memory usage. We also provide theoretical and architectural augmentations to source code transformation-based AD tools to perform FP error analysis. In this paper, we primarily focus on analyzing errors introduced by mixed-precision AC techniques, the most popular approximate technique in HPC. We also show the applicability of our tool in estimating other kinds of errors by evaluating our tool on codes that use approximate functions. Moreover, we demonstrate the speedups achieved by CHEF-FP during analysis time as compared to the existing state-of-the-art tool as a result of its ability to generate and insert approximation error estimate code directly into the derivative source. The generated code also becomes a candidate for better compiler optimizations contributing to lesser runtime performance overhead.

I. Introduction

As we enter the post-Moore era, where we no longer enjoy the free lunch of performance growth from shrinking the transistor features, researchers are exploring other computing paradigms to increase computational throughput. Approximate Computing (AC) has garnered significant interest as a promising approach for increasing peak performance. AC relies on the application to tolerate some amount of error to achieve performance gains. Among the various AC techniques that currently exist, reduced floating-point (FP) precision, or mixed precision, has gained in popularity. Computer architectures support multiple levels of precision for FP data and arithmetic operations — 64 bits *double* precision, 32 bits *single* precision, 128 bits *quad* precision, and 16 bits *half* precision. The choice of precision determines the amount of rounding error. While using higher precision for data and operations may result in increased accuracy, it can lead to an increase in application execution time, memory and energy consumption. Mixed-precision tuning involves using higher precision when necessary to maintain accuracy and using lower precision where we can improve performance.

Despite the availability of multiple levels of precision for FP, it is challenging to apply them in HPC and scientific codes. To use them effectively, developers need to understand the details of rounding errors as well as how they propagate through their applications. Due to the lack of scalable and rigorous tools that analyze error sensitivity in the applications' code regions, developers often resort to the safer option of using high precision throughout. We need scalable tools to understand the impact of lowering the precision of data and computation, identify error-tolerant regions, and provide guidance to users.

Several techniques have been proposed to estimate the sensitivity profile of an application, including automated search based approaches [1], [2], static analysis [3], [4], or using Automatic Differentiation (AD) [5]. Unfortunately, the existing set of tools fail to provide a feasible solution for HPC applications. Search-based approaches are very expensive as the state space is significantly large and quickly become infeasible even for small benchmarks. Static analysis-based approaches using interval analysis or Taylor series approximation provide rigorous estimates for FP errors but are so far limited to programs with a small number of operations [4]. Several methods [5], [6] leverage AD to determine error-resilient regions of codes. While they have been shown to work well for smaller HPC benchmarks, they often require manual code changes and incorporating several software tools together [7]. These tools are slow and have high memory overhead, making them infeasible for large HPC workloads.

We propose CHEF-FP, a scalable, flexible, and easy-to-

1530-2075/23/$31.00 ©2023 IEEE
DOI 10.1109/IPDPS54959.2023.00105

use tool, for analyzing approximation errors in HPC applications. We use Clad [8], an efficient AD tool built as a plugin to the Clang compiler as a backend for the presented framework and utilize its AD abilities to evaluate FP errors in C++ code. Clad builds derivatives by transforming the internal compiler representation of the program. This level of granularity allows CHEF-FP, built as an extension to Clad, to exploit a program's source information to automatically add auxiliary instructions to the generated derivative depending on a set of rules. Accordingly, CHEF-FP annotates code with FP error information without user intervention. This automation is beneficial for large-scale codes wherein going into hand-write annotations becomes complex, tedious, and error-prone. Moreover, CHEF-FP enables domain-specific FP error analysis due to the increased flexibility of a compiler-based backend. This allows easy modification of parts of the CHEF-FP framework to achieve a more tailored FP error analysis. This level of variability enables a finer-grained analysis where users can themselves query source information and make more precise decisions for the direction of the analysis.

Finally, because we generate FP error estimation (EE) code directly into the derivative source, we observe a significant speedup compared to other tools performing similar FP error analysis. The generated code also becomes a candidate for further compiler optimizations contributing to better runtime performance. We use the FP error profile provided by CHEF-FP to guide the design of a mixed-precision version of the code and achieve performance improvements of 8% for *HPCCG* and 65% for *Black-Scholes* while satisfying the user-specified error threshold. At analysis time, CHEF-FP obtained a maximum time speedup of 217% with the Simpsons benchmark and a memory efficiency of 632% with the Black-Scholes benchmark.

Key Contributions

- CHEF-FP, an efficient tool to automate AD-based FP error analysis. CHEF-FP inlines error calculations into the adjoint code and results in faster and more memory-efficient analysis, making it suitable for use in data-intensive applications;
- Formalism for augmenting AD to perform FP error analysis. Specifically, we present a generic way to extend source code transformation-based AD tools;
- Customizable EE module that supports any AD-based user-defined error model;
- Tool evaluation using a set of HPC benchmarks for mixed-precision and approximate function error analysis.

II. Background

Floating-point arithmetic operations are the most prevalent computation in HPC applications. Computer architectures support multiple levels of precision for FP data and arithmetic operations. In the IEEE 754 standard [9], the most common representation today for real numbers, choices are 64 bits *double* precision, 32 bits *single* precision, 128 bits *quad* precision, and 16 bits *half* precision. The choice of precision determines the amount of rounding error.

The FP rounding error is the accumulation of FP errors from each variable toward the target function's result. Target functions are modeled as a series of multiple assignment operations. These assignments and function inputs are assumed to be independent (FP error analysis on correlated variables is still a nascent field, and definitive methods of estimation are yet to be established). We define an expression to calculate and accumulate the FP error introduced by each assignment and refer to these error calculation and accumulation expressions as FP error models. CHEF-FP, aims to estimate an upper bound on these FP rounding errors.

A. Modelling Floating-Point Errors

The error model describes a metric that can be used in error estimation analysis. This metric is applied to all assignment operations in the function, and the results from its evaluation are accumulated into the total FP error of the left-hand side of the assignment. The Taylor series approximation is well suited to model floating point errors. Let's assume an arbitrary function $y = f(x)$, where x is represented in the standard IEEE 754 single precision. Assuming a floating point error of h in x, we define $\widetilde{f}(x) = f(x+h)$. A symbolic Taylor series expansion yields:

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

Here $f'(x)$ represents the derivative of the function with respect to x. Approximating the series to first-order results in:

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

An expression for the absolute floating point error in f is:

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

To determine the maximum absolute floating point error in f, we write $h = \epsilon_m|x|$ where ϵ_m is known as the *machine epsilon*. The machine epsilon gives the maximum relative representation error in floating point variables due to rounding. It is a machine-dependent value and follows the IEEE standard in most compilers. The absolute error can then be written as:

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

This error model is sufficient for most smaller cases and can produce loose upper bounds of the maximum permissible FP error in programs. Here, AD techniques can enable efficient computations of $f'(x)$ that scale to real-world workflows.

B. Automatic Differentiation Basics

AD takes as input program code that *has meaningful differentiable properties* and produces new code augmented with pushforward or pullback operators [10]. The AD mode, which produces pushforward operators to capture the sensitivity from inputs to outputs, is commonly known as *forward* mode. The AD mode capturing the sensitivity from outputs to inputs is called *reverse* mode, *adjoint* mode, or backpropagation. The computational merit of the pullback is that it provides a very efficient way to compute the function's gradient with relative time complexity which is independent of the input size [11].

AD tools can be categorized by how much work is done before program execution. AD typically starts by *building a computational graph*, or a directed acyclic graph of mathematical operations applied to an input. At one extreme, the *tracing*, or *taping* approach constructs and processes the computational graph at the time of execution each time a function is invoked. In contrast, the source transformation approach does as much as possible at compile time to create the derivative only once. The tracing approach is easier to implement and adopt into existing codebases. The source transformation approach has better performance but usually covers a subset of the language. **Tracing:** Records the linear sequence of computation operations at runtime into a *tape*. The control flow is flattened to produce a derivative. A typical implementation is via operator overloading, defining a special floating type with overloaded elementary operations. Algorithms use this type to trigger differentiation by calling a special function. There are numerous C++ AD tools based on tracing, including ADOL-C [12], and Adept [13]. As the derivative is produced at runtime, the just-in-time differentiation process is constrained to perform optimizations quickly. Metaprogramming techniques, such as expression templates, can mitigate the issue, but they cannot optimize across statements and generally do not handle control flow [13], [14]. **Source Transformation:** Constructs the computation graph and produces a derivative at compile time. More compile-time optimizations can be applied, such as reorganizing or evaluating simple constant expressions and common subexpression elimination. Source transformation is more difficult to implement as it requires a significant investment in developing and maintaining a language parser. Tapenade [15] is an example of a source transformation tool with custom parsers for C and Fortran. Source transformation tools usually do not support the full language feature set (e.g. certain language idioms are particularly hard to differentiate).

Historically, toolmakers made trade-offs between ease of use, performance, and ease of integration. AD now benefits from better language support to avoid such trade-offs. Recently, production compilers like Clang allowed tools to reuse the language parsing infrastructure. Enzyme [16] and Clad [8] are examples of compiler-based AD tools using such preexisting parsers.

III. AD-based FP Error estimation Using CHEF-FP

An important aspect of dealing with FP applications with high precision requirements is identifying the sensitive, or more error-prone, areas to devise suitable mitigation strategies. This is where AD-based sensitivity analysis can be very useful. It can find specific variables or regions of code with a high contribution to the overall FP error in the application.

The sensitivity of a variable x to FP errors (S_x) can be deduced from the default error model described in Eq. 1 as:

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

Here, the total contribution of FP errors to the function (either a routine or an entire program) by variable x increases as S_x increases. A study of the trends of these sensitivity values across the domain of a function can reveal insights into the numerical stability of the function and help determine possible causes of instabilities. Sensitivity values can also be used as a guide for a class of type-based optimizations called *Mixed Precision Tuning*.

Mixed Precision Tuning involves *demoting* certain variables to lower precision without severely affecting the overall accuracy of the application. In corollary, it involves preserving or *promoting* the precision of variables that have a significant effect on the accuracy. A mixed precision tuned configuration is only valid when the difference of the pre- and post-tuning accuracy is less than some defined threshold value. An effective way to maintain this requirement is by analyzing the sensitivity of all input and intermediate variables and selecting the ones with lower sensitivity to be demoted. The FP error contributions of the demoted variables are accumulated and compared to the threshold value. A mixed precision configuration is reached when the accumulated error meets the threshold value.

Taylor-based analyses described in section II-A require modeling assignments to FP variables and their respective adjoints. The adjoint accumulation mode naturally offers such mapping. The CHEF-FP implementation exports this information from Clad. Clad can be used either as a part of the compilation lowering pipeline or to generate source code that can be compiled by another compiler toolchain. Clad implements forward and adjoint mode AD, together with a flexible extension system that allows user code to subscribe to events during the process of adjoint creation.

Compiler-based AD tools can operate at the level of different program representations, and each implementation has its own set of pros and cons. For example, Clad implements AD on Clang's high-level representation to make use of better diagnostics, support compile-time programming, generate understandable source code, and use a standard optimization pipeline. These properties are key for AD-based FP analyses. If AD runs before the optimization pipeline, unsafe optimizations might introduce extra floating point errors. Running AD after optimization, as done by tools including Enzyme, avoids these errors. In this case, even standard optimizations could break differentiability. It is still to be seen if there is a way to combine the strengths of both approaches.

While many AD-based approaches exist for error analysis, they typically involve significant manual effort to perform code changes or long toolchains to automate it. In this work, we present CHEF-FP, an AD-based FP error estimation framework that requires less manual integration work and comes packaged with Clad, taking away the tedious task of setting up long toolchains. The tool's proximity to the compiler and the fact that the FP error annotations are built into the code's derivatives allows for powerful compiler optimizations that provide significant speedups of the analysis time when compared to the current state-of-the-art tools like ADAPT [5].

Another advantage of using source-level AD tools (like Clad) as the backend is that they provide important source insights. A higher-level representation, for example, enables us to identify and attribute FP errors to variables as they are visible at the source. Recovery of such information (variable names, IDs, source location, and so on) becomes difficult for lower-level tools, where optimizations can optimize away variables of interest. Clad can also identify special constructs (such as loops, lambdas, functions, if statements, and so on) and tune the error code generation accordingly. This information, while available to lower-level tools, is more difficult to extract and process at that level.

A. Source Transformation FP Error Estimation Using AD

Algorithm 1 describes a transformation that connects the variable-adjoint mapping available in a source transformation AD engine with an FP EE framework. For each function annotated for EE, we use the mapping (in *AdjointAD*) while differentiating (*NewFunction*) to hand the control to the EE model (*AssignError*) to insert extra instructions. In the end, we compute the total error (*FinalizeEE*) by allowing the custom model to pass it as an output parameter or print it on the screen. One important aspect is that the algorithm does not impose constraints on the structure of the right-hand side of the FP assignments. For implementations of certain functions, this implies that the right-hand side can contain arbitrary long expressions. In turn, the error contribution of longer expressions is computed with less precision. This feature gives users implicit control by allowing them to reduce the expression size manually. The generated code depends on the particular implementation of the EE model; however, we can schematically list a possible generalization for a better understanding of the process.

Algorithm 1 Error Estimation Generation Process

```
Require: Selected by $estimate\_error$ functions
Ensure: Generated error estimated functions ”$\overline{function}$”
for all $function$ in estimate_error.functions do
    $map\langle variable, adjoint\rangle \leftarrow$ AdjointAD($function$)
    $\overline{function} \leftarrow$ NewFunction($function, map$)
    for all $variable, adjoint$ in $map$ do
        AssignError($\overline{function}, variable, adjoint$)
    end for
    FinalizeEE($\overline{function}$)
end for
```

B. Sample Syntactic Structure of the Generated Function

Algorithm 1 transforms the input program in Fig 1 and produces another program as shown in Fig 2. Fig 1 represents a function written in a computer programming language that takes P parameters and returns a result of type T. Without loss of generality, its body consists of instructions denoted by $L_i, \forall i \in [1..n]$, where $S = f_i(S)$ manipulating internal state S and returning the final state S in $result(S)$. This notation does not exclude control flow constructs which can be represented with a sufficiently long linear sequence of L_i.

The adjoint accumulation mode of AD computes partial derivatives starting from the function's ($\overline{FuncName}$) outputs towards the function inputs (P). It requires executing the function in reverse order. Fig. 2 illustrates the AD adjoint accumulation transformation, which is a possibly augmented instruction sequence L_i. The augmented instruction records a subset of the internal state (denoted as $out(L_i) \subset S$, which depends on f_i) necessary to preserve the semantics when the instructions are executed in reverse order. A common implementation mechanism is to use a LIFO structure such as a *stack* and insert *push* or *pop* operations via *Push* or *Pop*. This transformation is known as *forward sweep*. The *backward sweep* is responsible for computing the adjoint ($\overline{S}$) for each instruction $\overleftarrow{L_i}$. $\overline{S}$ may require restoring altered state S by using a *pop* operation to correctly evaluate $f_i'(S)$.

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

Figure 1: Structure of the original function $FuncName$

```
function $\overline{FuncName}(P, E) : void$
        $Initialize\ S\ and\ \overline{S}$
$L_1$:        Push($out(L_1)$); $S = f_1(S)$                                    forward sweep
              ...
$L_i$:        Push($out(L_i)$); $S = f_i(S)$
              ...
$L_{n-1}$:    Push($out(L_{n-1})$); $S = f_{n-1}(S)$
$\overleftarrow{L_n}$:    $\overline{S} = \overline{S} \times f_n'(S)$; AssignError($\overleftarrow{L_n}$)    backward sweep
              Pop($out(L_{n-1})$)
              ...
              Pop($out(L_i)$)
$\overleftarrow{L_i}$:    $\overline{S} = \overline{S} \times f_i'(S)$; AssignError($\overleftarrow{L_i}$)
              ...
              Pop($out(L_1)$)
$\overleftarrow{L_1}$:    $\overline{S} = \overline{S} \times f_1'(S)$; AssignError($\overleftarrow{L_1}$)
        $E =$ FinalizeEE($\overline{FuncName}, \overline{S}$)
```

Figure 2: Structure of the error estimated function $\overline{FuncName}$

Our adjoint accumulation mode extension adds 3 elements: another output parameter E to $\overline{FuncName}$ modeling the total error; a callback call to *AssignError* for every assignment taking variable and its adjoint as parameters; and a function *FinalizeEE* computing the total error E.

C. Structural Operational Semantics of the Transformation

The transformation from the pseudo-code shown in Fig. 1 to its error estimation form, shown in Fig 2, can be described with the structural operational semantics notation. Similar to Hascoet and Pascual's work describing AD semantics in [15], we can extend the rules to describe the EE specific transformations. Due to space limitations, we show only essential rules supporting FP EE. They describe how a generic source-transformation AD framework can be extended to support AD-based FP analysis. The rest of the rules associated with AD are already available in [15].

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

In rule S1, the premises are fulfilled when: (i) Predicate *isEstErrFunction* is true when *FuncName* is selected for being an error estimation candidate; (ii) *newFunction* can create a $\overline{FuncName}$; (iii) *Params* and *Locals* can be transformed; and (iv) The original function's statement sequence has been transformed into a forward and backward sweep. When all antecedents are fulfilled, *FinalizeEE* is added at the end of the backward sweep.

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

In rule S2, the premises are fulfilled when: (i) *isLive* is true when the statement *Stmt* is useful for the derivative computation; (ii) *isDiff* is true when the memory location is relevant for the derivative computation; (iii) *newLocal* can create a variable of the type of *Ref*; (iv) *Expr* can differentiate *Expr*; and (v) *buildAssignError* can generate EE instructions *Est*. When all antecedents are fulfilled, then *AssignError* augments the backward sweep. *Ref* represents a memory location where a value can be stored.

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

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

The next two rules S3 and S4 consider the variations of the values of the *isLive* and *isDiff* predicates.

When either *isDiff* or *isLive* is false, then we still create the adjoints and insert the *AssignError* to capture variable's error contribution. Function calls and parameter passing can be expressed by analogy.

D. Programming Model & Framework Design

Clad's callback system allows the creation of extensions that can augment generated code. We use this ability to build a lightweight framework to insert FP error estimation code in Clad generated adjoints. CHEF-FP leverages the flexible design of Clad and adds itself as a native extension that synthesizes error estimation code as part of the differentiation process. Fig. 3 outlines its high-level design. CHEF-FP's implementation is broadly divided into an *Error Estimation Module* and an *Error Model*.

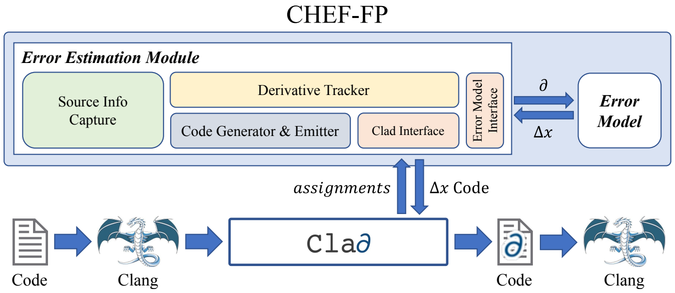

Figure 3: **FP EE Generation Workflow in CHEF-FP**. The control from Clang's compilation pipeline is intercepted by Clad, and detection of calls to `estimate_error()` instantiates the *Error Estimation Module* and sets up callbacks. The *Error Estimation Module* then listens to callbacks from Clad's adjoint mode and is responsible for augmenting the derivative body with the error estimation code defined in the *Error Model* before passing the control back to Clad. The *Error Estimation Module* is also responsible for caching values, tracking derivatives, capturing the required structural and source information, and accumulating errors.

Error Estimation Module: The error estimation module is the major component that makes up CHEF-FP. The registration of calls to `estimate_error()`, by which the user annotates the functions of interest, instantiates the *Error Estimation Module*. This sets up error estimation callbacks and passes the control back to Clad. The *Error Estimation Module* listens to callbacks from Clad's adjoint mode and takes control when an interesting (from EE perspective) callback is triggered. The *Error Estimation Module* augments the derivative body by adding the EE expressions received from the *Error Model* and passes control back to Clad until the next callback is invoked. Thus, it is responsible for interfacing with Clad, defining the various methods that facilitate the generation and emission of FP error estimation code, and exchanging information with the *Error Model* to produce specifically augmented code. The error estimation module also caches values, tracks derivatives, captures the required structural and source information, and accumulates errors to provide the final FP error estimate.

Error Model: The Error Model defines an interface to describe the error expression to generate. It receives derivative expressions as the derivative is being generated and returns respective error expressions. By default, a call to a user-defined function `getErrorVal` is synthesized, allowing users to write their formulas in plain C/C++ code. An additional interface that can be programmed to generate advanced calls that export more adjoint information to incorporate other advanced custom *Error Model*s also exists. This interface can be completely customized to make the FP error analysis more specific to an application. Listing 1 demonstrates the programming model with a minimal example.

Listing 1 Minimal demonstrator of the usage of CHEF-FP.

```
float func(float x, float y) {
  float z;
  z = x + y;
  return z;
}
int main() {
  // Call estimate error on target function.
  auto df = clad::estimate_error(func);
  // Declare the inputs, their derivative
  // outputs and the final error output.
  float x = 1.95e-5, y = 1.37e-7;
  float dx = 0, dy = 0;
  double fp_error = 0;
  // Execute the generated code.
  df.execute(x, y, &dx, &dy, fp_error);
  // fp_error now contains the error of func.
  std::cout << "Error in func: " << fp_error;
}
```

E. Implementation

CHEF-FP registers an API (`estimate_error`) in Clad that can be used to calculate the floating point errors in a given function using a default error estimation model. An example of a typical invocation of CHEF-FP is illustrated in listing 1.

For more complex analyses, it may be necessary to change the underlying error model to achieve satisfactory estimates. This can simply be achieved by implementing the `FPErrorEstimationModel` interface with the appropriate error model, compiling it into a shared library, and passing that library to CHEF-FP. An example of such an implementation is described in listing 2. Here `AssignError` builds the code expression (expressed as Clang's internal expression types) that can then be emitted into code by the error estimation module. This function exposes three values that can be used to implement the custom model - the name of the variable, the variable itself, and its derivative.

Listing 2 A template for declaring custom model classes.

```
struct CustomModel : public
↪  FPErrorEstimationModel {
  CustomModel(DerivativeBuilder& builder)
      : FPErrorEstimationModel(builder) {}
  // Returns the error expression to be
  // calculated for each variable assignment.
  clang::Expr* AssignError(StmtDiff refExpr,
   ↪  const char* name) override;
};
```

Listing 3 An example implementation of `AssignError` that builds calls to external functions. The implementation generates a call to a user-defined function `getErrorVal`.

```
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) {
  // Build a vector-like container to store
  // the parameters of the function call.
  llvm::SmallVector<clang::Expr*, 3> params{
      refExpr.getExpr_dx(), refExpr.getExpr(),
       ↪  utils::CreateStringLiteral(
       ↪  m_Context, name)};
  // Return a call to getErrorVal.
  return GetFunctionCall("getErrorVal",
   ↪  "clad", params);
}
```

There are two broad ways to implement `AssignError` - building an arithmetic expression or building an external function call. Both methods involve building an assignable expression. The former approach involves directly building the arithmetic expression to be assigned and so it is fairly limited. It also requires that the model be re-built for every modification in `AssignError`. While we provide an API useful for building simple expressions, it becomes unintuitive to implement complex models based on just a single expression. Hence, we build calls to external functions as a valid error model as long as the function has a compatible return type to the variable being assigned the error. This approach allows users to define their error models as regular C++ functions, allowing for the implementation of more computationally complicated models.

Listing 3 demonstrates how to implement an arbitrary function `getErrorVal` and use it as a custom error model. In the listing, we build the model used in ADAPT-FP, whose mathematical notation is described as follows:

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

where Δ is the accumulated error in the function f due to the rounding errors for all x_i. Here, the error assigned to each variable is the difference between its single and double precision values. Note, this model imposes a requirement that all functions it is used on are of double or higher precision.

It is also possible to modify the signature and composition of the example external error function (i.e., `getErrorVal`) defined in listing 3 by modifying the `AssignError` function. This allows users to add relevant meta information of the variables exposed by Clang. For example, users can implement `AssignError` to build calls to an error function that can take in a variable's source location. This information can be used for identifying specific areas of code more prone to errors.

IV. Experiments

We compare CHEF-FP against the current state-of-the-art ADAPT using five different algorithms: *Arc Length*, *Simpsons*, *k-Means clustering*, *HPCCG*, and *Black-Scholes*. For the first 4 benchmarks, we use the error model described in equation 2. The time taken is measured using *Google benchmark* and peak memory by *GNU time*. The benchmarks were done on the Princeton *Tiger* cluster [17] with a 2.4GHz Intel Xeon Gold 6148 CPU and 188 GB of RAM. Similar benchmark results were obtained using the LLNL Quartz [18] system, which is a cluster consisting of Intel Xeon E5-2695 processors with 2.1 GHz cores and 128 GB of memory per node.

Benchmark	**Threshold**	**Actual Error**	**Estimated Error**	**Speedup**
Arc Length	1e-05	3.24e-06	3.24e-06	1.11
Simpsons	1e-06	7.80e-08	1.32e-07	2.25
k-Means	1e-06	0.00e+00	0.00e+00	-
HPCCG	1e-10	5.21e-12	5.92e-11	1.08

Table I: **Error and performance measurements of the mixed precision versions of the benchmarks**. The table shows a comparison between the actual error and CHEF-FP's estimated error in the mixed precision versions of the original program. It also shows the execution speedup of the mixed variant. For k-*Means*, CHEF-FP's identified mixed precision configuration for the defined threshold showed no speedup.

For all of the following benchmarks, we present reductions in the analysis time and memory while still producing results that are at par with tools such as ADAPT. We also utilize this section to comment on how the analysis for certain examples can be extended and how CHEF-FP can be used to perform more sophisticated approximate analysis. Lastly, we present a summary of the mixed precision analysis results in table I to demonstrate how CHEF-FP can produce correct floating-point analysis results while requiring little to no user intervention.

1) Arc Length: The arc length function approximates a curve's length (L) by sampling various points on the curve and summing up the straight line distance between two consecutive points on the curve.

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

We vary the number of iterations, n, that the arclength algorithm is run for, to benchmark CHEF-FP against ADAPT. Fig. 4 compares the time taken and memory used to analyze the algorithm for mixed precision analysis. The absence of a data point for ADAPT at 10^8 iterations is due to it running out of memory. CHEF-FP's memory footprint is lower because the error calculation is inlined in the gradient function.

2) Simpsons: Simpsons is an iterative algorithm to approximate the integral of a function in the given interval by summing the integral over multiple small intervals:

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

Here, $f_i = f(a + ih)$, $h = (a + b)/2n$, and n is the number of iterations. Fig. 5 compares ADAPT and CHEF-FP on the basis of time and memory usage. Similar to arc length we vary the number of iterations to benchmark the two tools. CHEF-FP's recommended mixed precision configuration gives a speedup of 2.25 times when compared against the same program in higher precision. It is also able to predict the actual error in the mixed-precision version of the application, as seen in table I.

3) k-Means Clustering: Part of the Rodinia benchmark suite [19], the k-Means clustering algorithm is used for grouping multiple data points into k clusters. We instrument the Euclidean distance function as it is the major computational hotspot of the application. Similar to previous benchmarks, we compare the performance of CHEF-FP against ADAPT in fig. 6.

The Euclidean distance function has three major variables: *attributes*, *clusters*, and *sum*. It can be represented as follows:

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

The error estimated by Clad for *attributes* is 0 because the input data of the benchmark is represented with four digits after the decimal. The errors estimated for *clusters* and *sum* are higher than the threshold set for k-Means in table I. Hence, CHEF-FP recommends only converting *attributes* to lower precision. To test out CHEF-FP's error estimates, we went a step ahead and converted each of the variables to lower precision individually and found the actual error introduced by them. These configurations were executed on 10^6 datapoints, and the findings are shown in table III.

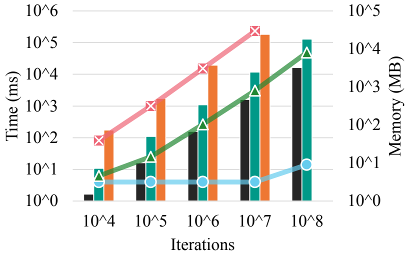

Figure 4: Arc Length

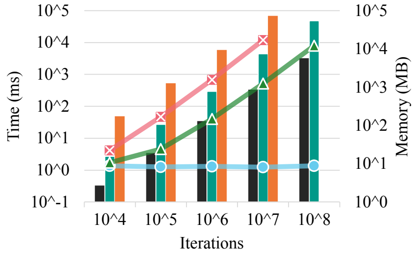

Figure 5: Simpsons

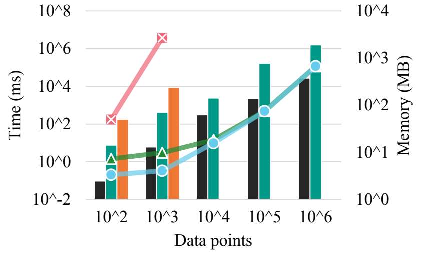

Figure 6: k-Means algorithm

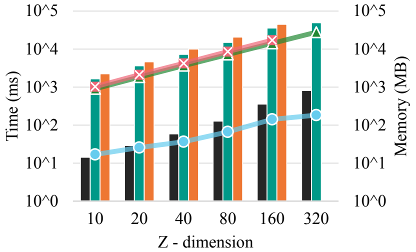

Figure 7: HPCCG

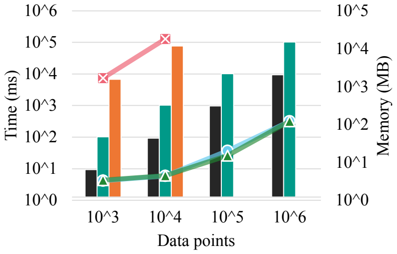

Figure 8: Black-Scholes

Benchmark	Time	Memory
Arc length	1.61x	1.95x
Simpsons	2.17x	1.44x
k-Means	2.02x	4.44x
HPCCG	1.03x	1.02x
Black-Scholes	1.76x	6.32x

Table II: Performance Improvements

Figures 4 to 8 show the results of benchmarking CHEF-FP, ADAPT, and the original function. The labels below the function show the algorithm being benchmarked. The lines show peak memory usage and the bars represent the time taken during the FP error analysis of the given algorithm. **Table II summarises CHEF-FP's performance improvements over ADAPT.** The improvements are given as 'times improved' over the FP error analysis and represent the average improvement across all the data points. Our benchmarks show that CHEF-FP outperforms ADAPT, which is the current state-of-the-art AD-based FP error estimation tool while producing mixed precision analysis results that agree with ADAPT's analysis.

Variable(s) in Lower Precision	Actual Error	Estimated Error
attributes	00e+00	00e+00
clusters	3.67e-04	9.35e-04
sum	8.33e-04	7.08e-03
all 3	2.40e-03	8.01e-03

Table III: **k-Means – Error measurements of various mixed precision configurations**. We demote the 3 variables (*attributes*, *clusters* and *sum*) to lower precision one by one and compare the resulting errors with CHEF-FP's estimate.

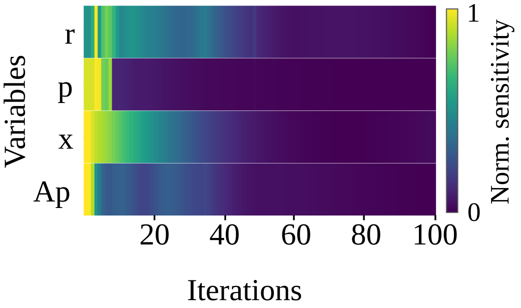

Figure 9: **HPCCG Variable Heatmap.** This illustrates the normalized sensitivity of the variables r, p, x, and Ap for every iteration. This sensitivity profile can be used to determine more fine-tuned optimizations.

4) HPCCG: Part of the Mantevo benchmark suite, HPCGG is a simple conjugate gradient benchmark code for a 3D chimney domain converted to be single-threaded. The analysis is done while scaling the inputs from the base dimension of $20 \times 30 \times 10$ to the recommended size of $20 \times 30 \times 160$ and then further to $20 \times 30 \times 320$. The results in fig. 7 show that ADAPT runs out of memory for the $20 \times 30 \times 320$ dimension.

CHEF-FP is also used to analyze HPCCG for a possible loop perforation based optimization. We already generated errors for intermediate variables, so we only needed to slightly tweak it to dump the sensitivity of the variables over each iteration. We analyzed the change in sensitivity of the various variables over the total run of the application, and we found that the sensitivity for all variables drops below our set threshold after almost 60 iterations. The normalized sensitivity of the variables is shown in the form of a heat map in fig. 9. Based on these findings, we split the main loop of HPCCG into two chunks - the first chunk runs the full loop for the first 60 iterations in high precision, and the second chunk runs for the remaining iterations in lower precision. This configuration gives us a speedup of 8% as shown in table I.

5) Black-Scholes: Part of the Parsec-benchmark suite [20], the Black-Scholes equation is a differential equation that describes the change in the value of an option as the price of the underlying asset changes. To compare against ADAPT, we analyze the function that calculates the option price depending on the factors that influence the stock. ADAPT is not able to scale beyond 10^4 data points in our benchmarks. The results are shown in fig. 8. For this benchmark, both CHEF-FP and ADAPT showed that most of the intermediate variables are very sensitive to errors and could not identify an effective mixed precision configuration.

In addition, we leveraged the customizability of CHEF-FP to analyze the effect of using Paul Minero's FastApprox library [21] instead of the standard C math library to gain performance improvements. The FastApprox library provides approximate versions of various math functions that trade off accuracy for performance. CHEF-FP can determine the error introduced by replacing the standard versions of the math functions with the approximate ones in any program. We identified three math functions in the Black-Scholes application that had approximate versions in the FastApprox library. The inputs to these functions were identified, and a map was formed with their names mapping to the functions they were input to. The map is used in a custom model to match each variable with the correct function and its approximation. These values are then used by the custom model to accurately estimate the errors due to approximation in the application. The algorithm for the custom model is illustrated in algorithm 2. The estimated error, actual error, and speedup from FastApprox in the Black-Scholes application are shown in table IV.

Algorithm 2 Estimating approximation-based errors.

```
Require: input variable as $x$ and its name as $name$, the partial
    derivative of $x$ wrt. the function as $dx$, and a map of
    variables of interest as $S : name \rightarrow$ function name
$\Delta \leftarrow 0$
if $name$ is contained in $S$ then
    $fName \leftarrow$ S.GETVALUE($name$)
    $\Delta \leftarrow$ EVAL($fName$, $x$)$-$EVALAPPROX($fName$, $x$)
    $\Delta \leftarrow \Delta \div (\partial$ EVAL($fName$, $x$) $/ \ \partial x)$
end if
$xApproxError \leftarrow |dx * \Delta|$
REGISTERERROR($name$, $xApproxError$)
return $xApproxError$
```

V. Discussion

CHEF-FP shows consistently better performance than ADAPT while suggesting similar hints for mixed precision tuning and yielding similar performance benefits. CHEF-FP provides an efficient, straightforward, and flexible way to analyze FP errors in complex C++ applications. We demonstrate the use of CHEF-FP for mixed precision tuning and sensitivity analysis on 4 benchmarks – *Arc Length*, *Simpsons*, *k-Means* and *HPCCG*, and its use for approximation analysis on 1 benchmark – *Black-Scholes*. In this section, we discuss in more detail the experimental results and current limitations.

A. *Summary of Results From* CHEF-FP

CHEF-FP identified stable mixed precision configurations for the *Arc Length* and *Simpsons* benchmarks for a given threshold; on conversion to these mixed precision configurations, both applications saw a noticeable speedup, summarised in table I. For the *k-Means* benchmark, CHEF-FP could not identify a mixed precision configuration (with a threshold of 10^{-6}) that resulted in a speedup. However, CHEF-FP provided good upper-bound estimates of the error in the application for different mixed precision configurations (table III).

For the *HPCCG* benchmark, the tool was used to analyze the sensitivity of variables across the main loop. This allowed for us to discover a mixed precision configuration that involved splitting the main loop into performing the first 60 iterations in higher precision and the rest in lower precision. Lastly, we also demonstrated the flexibility of CHEF-FP by leveraging its custom model support to perform an approximation error analysis on the *Black-Scholes* benchmark. CHEF-FP was able to accurately quantify the errors related to approximation in a set of specific functions. Additionally, CHEF-FP generated two different approximation configurations, and a report on the estimated errors from the same is shown in table IV.

B. *Current Limitations*

Quantifying overhead of type-casts: It is possible for a mixed-precision configuration to show worse performance than the high-precision version. Usually, this is due to the overhead of the implicit type-casts that have been introduced in the code by lowering the precision of some variables. One way to combat these overheads is to keep track of them via counters; for example, a trivial implicit casts counter can be implemented using Clang's AST Matchers.

Variety of analysis datasets: The results of the mixed-precision configurations formed here are input dependent. To form a general mixed-precision configuration, it is important to analyze the application over a representative set of inputs.

Source rewriting for mixed precision configurations: CHEF-FP provides sensitivity profiles, and error estimates to guide the process of mixed-precision re-implementation. Currently, we manually rewrite the source code to implement the mixed precision configurations suggested by CHEF-FP. We can use source transformation tools, such as Typeforge [22], to automate the generation of mixed-precision code. As future work, this process can be automated by combining the decision-making and code generation by using the error information at runtime to just-in-time optimize areas of code with lower sensitivity to run in lower precision.

Compiler optimizations: Certain floating-point unsafe optimizations (such as *--ffast-math* or *-fp-model fast*) can cause CHEF-FP's predicted errors to be different than the actual errors. Currently, CHEF-FP cannot distinguish errors introduced by optimizations from the expected FP errors. Since these optimizations take place post the derivative generation, certain substitution optimizations may cause even the derivative to be incorrect, leading to the underlying error propagation to also be incorrect. Users of CHEF-FP should be careful as these optimizations can cause incorrect analysis results. Another way the analysis results can be affected is by changing the intermediate rounding mode for mixed precision expressions (through flags such as *-fp-model*). Changing this may cause

App Configuration	Actual Errors			Estimated Errors			Speedup
	avg.	*max.*	*acc.*	*avg.*	*max.*	*acc.*	
FastApprox w/o Fast exp	1.16e-04	9.16e-04	1.16e+01	1.16e-04	9.16e-04	1.16e+01	1.14
FastApprox w/ Fast exp	5.88e-04	3.49e-03	5.88e+01	1.18e-03	3.25e-03	1.18e+02	1.65

Table IV: **Black-Scholes - Error and performance analysis of the various FastApprox based configurations**. This table shows the actual error and CHEF-FP's estimated error of the approximate version of the application. More specifically, it outlines the average, maximum, and accumulated error over 1000 data points. For the first row, the original program uses approximate versions of the *log* and *sqrt* functions. For the second row, the application uses the approximate version of *exp*.

the mixed precision version of the program to suffer performance degradation. We recommend using the *source* mode for rounding of intermediate calculations for consistent results.

VI. Related work

Many floating point error analysis tools have been proposed in literature, including both static and dynamic techniques. Dynamic approaches require running the program to gather necessary information to perform analysis. Brown et al. [23] designed FloatWatch, built on Valgrind, to determine if floating-point operations can be optimized by using a lower precision representation or fixed-point arithmetic. This is done by tracking the maximum difference between single and double precision computations by performing both simultaneously. Benz et al. [24] presented an approach where every floating-point computation is executed side by side in higher precision to assist the programmer in locating floating-point accuracy problems. Lam et al. [25] proposed a dynamic approach using a binary analysis tool, DynInst, to detect floating-point cancellations. An et al. [26] developed a dynamic binary analysis, FPInst, based on DynInst to compute errors by applying simple error accumulation formulas and tracking the error throughout a program. Static analysis tools, such as FPTaylor [4], SATIRE [27], and Precisa [28], use a global optimizer to estimate the upper bound of rounding errors. Gappa [29] automates error evaluation and propagation using interval arithmetic. Static analysis approaches provide a rigorous error analysis, but have been applied only to small benchmarks. SEESAW [3] employs symbolic adjoint mode AD to give tighter error bounds for intervals of input values and has been shown to work on practical HPC benchmarks. None of these methods targeted mixed-precision.

Several efforts have evaluated whether a program can take advantage of mixed-precision. Most of the techniques used search-based optimization to select suitable mixed-precision versions of the program that satisfies a user-provided error threshold. Lam et. al [28], [30] proposed CRAFT which uses a search algorithm to automate the identification of code regions that can use lower precision. Gonzales et al. [1] used delta debugging to narrow the search space for mixed-precision configuration. It was extended to consider groups of variables to further reduce the search space [31]. Laguna et. al [32] proposed GPUMixer, a tool used to tune FP precision on GPU programs with a focus on performance improvements. It uses shadow computations analysis to compute the error introduced by mixed-precision for each kernel and uses a search-based technique to identify the best mixed-precision configuration. These methods work by identifying a set of variables that can be in single precision while leaving the rest of the variables in double precision. Search-based techniques have the drawback that they require several runs of the application, and exploring the space is extremely time-consuming.

There have been several efforts directed towards analyzing applications for introducing mixed precision as well as estimating the error due to reduced precision representation using AD. AD has been used for estimating the rounding error in numerical algorithms since the early 90s [33], where partial derivatives given by AD were used to obtain a first-order approximation of the global rounding error due to elementary rounding errors. Interval analysis with the mean value theorem was used to provide tighter upper-bound for rounding error estimation. Later Langlois [34] proposed the CENA method, where a correction term was introduced to the first-order effect of rounding errors on the output of the numerical algorithms to improve the accuracy of estimation. Subsequently, ADAPT [5] used AD to estimate errors, enabling mixed-precision tuning by identifying regions where lower precision can be applied while staying within an error threshold. While ADAPT provided guidance for mixed-precision implementation, it involved manual annotations and code transformations. To address this issue, FloatSmith [7] was introduced, it integrated ADAPT with Codipack (AD tool) [35] and Typeforge (based on Rose [36] compiler) to automate the process of analyzing numerical codes. However, this long toolchain made it slow and cumbersome to work with.

VII. Conclusion

In this paper, we have defined formalism to augment automatic differentiation to perform floating-point error analysis, and demonstrated an efficient tool using compiler-based source transformation that does not overwhelm the already overly complex reverse accumulation AD mode. We present CHEF-FP, a flexible, scalable, and easy-to-use source-code transformation AD-based tool for the analysis of approximation errors in HPC applications. CHEF-FP works on the source level to inject error estimation code into generated adjoints, allowing analysis to be sped up via compiler optimizations. This setup allows CHEF-FP to operate on higher memory loads when compared to other FP error estimation tools. It provides considerable flexibility on what estimation code is generated by using custom error models, facilitating the exploration of other areas of approximation error analysis.

We demonstrated that CHEF-FP performs the same analysis as ADAPT-FP in a time and memory-efficient manner using five benchmarks. At analysis time, CHEF-FP obtained a maximum speedup of 2.17x over ADAPT-FP for the Simpsons benchmark and a memory reduction of 6.32x for the Black-Scholes benchmark. We showed how CHEF-FP could be used to perform sensitivity analysis and further provided recommendations on how to perform mixed-precision tuning on various applications. We illustrated how it could be used to accurately evaluate different precision configurations. Finally, we explored estimating approximation-based errors and evaluated the resulting approximate configurations on the *Black-Scholes* benchmark showing a speedup of up to 65%.

The open-source artifact for this work is available at the DOI: 10.5281/zenodo.7660443.

VIII. Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions. This project is supported by National Science Foundation under Grant OAC-1931408 and under Cooperative Agreement OAC-1836650. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research and by the LLNL-LDRD Program under Project No. 20-ERD-043. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-841475).

References

[1] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning assistant for floating-point precision," in *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2013, pp. 1–12.

[2] M. O. Lam and J. K. Hollingsworth, "Fine-grained floating-point precision analysis," *The International Journal of High Performance Computing Applications*, vol. 32, no. 2, pp. 231–245, 2018.

[3] A. Das, T. Tirpankar, G. Gopalakrishnan, and S. Krishnamoorthy, "Robustness analysis of loop-free floating-point programs via symbolic automatic differentiation," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2021, pp. 481–491.

[4] A. Solovyev, M. S. Baranowski, I. Briggs, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan, "Rigorous estimation of floating-point round-off errors with symbolic taylor expansions," *ACM Trans. on Programming Languages and Systems (TOPLAS)*, vol. 41, no. 1, pp. 1–39, 2018.

[5] H. Menon, M. O. Lam, D. Osei-Kuffuor, M. Schordan, S. Lloyd, K. Mohror, and J. Hittinger, "Adapt: Algorithmic differentiation applied to floating-point precision tuning," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 614–626.

[6] V. Vassiliadis, J. Riehme, J. Deussen, K. Parasyris, C. D. Antonopoulos, N. Bellas, S. Lalis, and U. Naumann, "Towards automatic significance analysis for approximate computing," in *2016 IEEE/ACM Int. Symposium on Code Generation and Optimization*. IEEE, 2016, pp. 182–193.

[7] M. O. Lam, T. Vanderbruggen, H. Menon, and M. Schordan, "Tool integration for source-level mixed precision," in *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness)*, 2019, pp. 27–35.

[8] V. Vassilev, M. Vassilev, A. Penev, L. Moneta, and V. Ilieva, "Clad — Automatic Differentiation Using Clang and LLVM," *Journal of Physics: Conference Series*, vol. 608, no. 1, p. 012055, 2015, [Link].

[9] "Ieee standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.

[10] M. Betancourt, "A geometric theory of higher-order automatic differentiation," *arXiv preprint arXiv:1812.11592*, 2018.

[11] A. Griewank and A. Walther, *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[12] A. Griewank, D. Juedes, and J. Utke, "Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++," *ACM Transactions on Mathematical Software (TOMS)*, vol. 22, no. 2, pp. 131–167, 1996.

[13] R. J. Hogan, "Fast reverse-mode automatic differentiation using expression templates in C++," *ACM Transactions on Mathematical Software (TOMS)*, vol. 40, no. 4, pp. 1–16, 2014.

[14] M. Sagebaum, T. Albring, and N. R. Gauger, "Expression templates for primal value taping in the reverse mode of algorithmic differentiation," *Opt. Methods and Software*, vol. 33, no. 4-6, pp. 1207–1231, 2018.

[15] L. Hascoet and V. Pascual, "The Tapenade automatic differentiation tool: principles, model, and specification," *ACM Transactions on Mathematical Software (TOMS)*, vol. 39, no. 3, pp. 1–43, 2013.

[16] W. S. Moses and V. Churavy, "Instead of Rewriting Foreign Code for ML, Automatically Synthesize Fast Gradients," in *Advances in Neural Information Processing Systems 33*. Curran Associates, Inc., 2020.

[17] "Princeton Research Computing," [Link], accessed 2023-02-20.

[18] "Livermore Computing: HPC at LLNL," [Link], accessed 2023-02-20.

[19] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[20] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[21] P. Mineiro, "Fastapprox," [Google Archive Link], 2011.

[22] N. T. Pinnow, M. Schordan, and T. L. Vanderbrugger, "Typeforge," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2019.

[23] A. W. Brown, P. H. Kelly, and W. Luk, "Profiling floating point value ranges for reconfigurable implementation," in *Proceedings of the 1st HiPEAC Workshop on Reconfigurable Computing*, 2007, pp. 6–16.

[24] F. Benz, A. Hildebrandt, and S. Hack, "A dynamic program analysis to find floating-point accuracy problems," *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 453–462, 2012.

[25] M. O. Lam, J. K. Hollingsworth, and G. Stewart, "Dynamic floating-point cancellation detection," *Parallel Computing*, vol. 39, no. 3, pp. 146–155, 2013.

[26] D. An, R. Blue, M. Lam, S. Piper, and G. Stoker, "Fpinst: Floating point error analysis using dyninst," 2008.

[27] A. Das, I. Briggs, G. Gopalakrishnan, S. Krishnamoorthy, and P. Panchekha, "Scalable yet rigorous floating-point error analysis," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–14.

[28] M. A. Feliú, M. Moscato, C. A. Muñoz *et al.*, "An abstract interpretation framework for the round-off error analysis of floating-point programs," in *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 2018, pp. 516–537.

[29] M. Daumas and G. Melquiond, "Certification of bounds on expressions involving rounded operators," *ACM Transactions on Mathematical Software (TOMS)*, vol. 37, no. 1, pp. 1–20, 2010.

[30] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. LeGendre, "Automatically adapting programs for mixed-precision floating-point computation," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, 2013, pp. 369–378.

[31] H. Guo and C. Rubio-González, "Exploiting community structure for floating-point precision tuning," in *Proc. of the 27th ACM SIGSOFT Int. Symposium on Software Testing and Analysis*, 2018, pp. 333–343.

[32] I. Laguna, P. C. Wood *et al.*, "Gpumixer: Performance-driven floating-point tuning for gpu scientific applications," in *Int. Conf. on High Performance Computing*. Springer, 2019, pp. 227–246.

[33] M. Iri, "History of automatic differentiation and rounding error estimation," *Andreas Griewank and George Corliss, editors*, pp. 3–16, 1991.

[34] P. Langlois, "A revised presentation of the cena method," Ph.D. dissertation, INRIA, 2000.

[35] N. G. M. Sagebaum, T. Albring, "High-performance derivative computations using codipack," *ACM Transactions on Mathematical Software (TOMS)*, vol. 45, no. 4, 2019.

[36] D. Quinlan and C. Liao, "The ROSE source-to-source compiler infrastructure," in *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, vol. 2011. Citeseer, 2011, p. 1.