ADAPT: Algorithmic Differentiation Applied to Floating-Point Precision Tuning

Harshitha Menon[*], Michael O. Lam[†], Daniel Osei-Kuffuor[*], Markus Schordan[*],
Scott Lloyd[*], Kathryn Mohror[*], Jeffrey Hittinger[*]
[*]Center for Applied Scientific Computing,
Lawrence Livermore National Laboratory
Email: {harshitha,oseikuffuor1,schordan1,lloyd23,kathryn,hittinger1}@llnl.gov
[†]James Madison University
Email: lam2mo@jmu.edu

***Abstract*—HPC applications use floating point arithmetic operations extensively to solve computational problems. Mixed-precision computing seeks to use the lowest precision data type that is sufficient to achieve a desired accuracy, improving performance and reducing power consumption. Manually optimizing a program to use mixed precision is challenging as it not only requires extensive knowledge about the numerical behavior of the algorithm but also estimates of the rounding errors. In this work, we present ADAPT, a scalable approach for mixed-precision analysis on HPC workloads using algorithmic differentiation to provide accurate estimates about the final output error. ADAPT provides a floating-point precision sensitivity profile while incurring an overhead of only a constant multiple of the original computation irrespective of the number of variables analyzed. The sensitivity profile can be used to make algorithmic choices and to develop mixed-precision configurations of a program. We evaluate ADAPT on six benchmarks and a proxy application (LULESH) and show that we are able to achieve a speedup of 1.2x on the proxy application.**

I. Introduction

Floating-point arithmetic remains the predominant means for real-valued computation in high-performance computing. As computation scales to exascale and beyond, the challenges of using floating-point arithmetic effectively will only increase [1], [2]. Computer architectures support multiple levels of precision for floating-point data and arithmetic operations. The standard IEEE floating-point precision choices for computation are 32 bits (*single-precision*), 64 bits (*double-precision*), 128 bits (*quad-precision*, usually implemented in software), as well as 16 bits (*half-precision*), which is now available on certain platforms such as NVIDIA GPUs. The choice of floating-point precision determines the magnitude of rounding errors in the computation.

Although a higher precision may improve the accuracy of the program output, it usually results in an increase of application run time, energy consumption, memory pressure, and interconnect usage. Often, programmers resort to the safer option of uniformly applying higher precision (e.g., IEEE *double precision*) throughout the program to ensure accuracy of the simulation output at the expense of performance. However, not all applications require higher precision, and for some applications, their precision requirement is dependent on the input. Ideally, applications should use the lowest precision necessary to achieve the required accuracy in order to take advantage of the performance and energy benefits of using the lower precision.

One promising approach is the use of mixed- or variable-precision arithmetic, i.e., using multiple levels of precision in a single program, gaining the benefits of high-precision arithmetic where necessary to maintain accuracy but using lower-precision arithmetic where possible to improve performance [3] and reduce energy usage [4]. However, developers must take care to ensure that the errors introduced are within the acceptable thresholds so as not to corrupt the output; large errors can render results useless. Unfortunately, it is not always easy to develop mixed-precision versions of large code bases manually, as this not only requires extensive knowledge of the numerical behavior of the algorithm but also requires understanding of the subtle details of floating-point rounding errors. This process becomes increasingly infeasible for larger-scale HPC programs with multiple modules.

There have been many efforts to automate this process in various ways [5], [6], [7], [8], [9], [10]. Some of this work involves automatically discovering unstable floating-point executions [11], [12], [13] and accuracy-improving transformations based on a database of rewrite rules [10], [9]. However, not all of these error-introducing operations propagate to the final output. Dynamic automated search based approaches [5], [6] evaluate different mixed-precision configurations of the program to identify the best configuration that satisfies the error threshold. The main drawback of these approaches is that the state space to explore is exponential in the number of variables; exploring even a subset is very time-intensive. The search-based algorithms can also get trapped in local minima. Various static analysis approaches [14], [7] use interval and affine arithmetic or Taylor series approximations to analyze stability and to provide rigorous bounds on rounding errors. However, they do not scale very well and thus have not been applied to HPC workloads. See Section VII for more discussion of these existing tools.

We propose an approach, implemented in a tool called ADAPT (Algorithmic Differentiation Applied to Precision Tuning), that uses algorithmic differentiation [15] (a method for numerically computing the derivative of computer pro-

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00 ©2018 IEEE

grams) to analyze floating-point precision sensitivity of variables and operations in a program. We then use the results to develop a mixed-precision version of the program. Our techniques enable the scaling of rigorous precision analysis techniques to benchmarks and proxy applications, which is an important step toward the application of mixed-precision analysis to full-scale HPC applications and the realization of the many benefits mentioned earlier.

The primary contributions of this paper are as follows:

- ADAPT, an approach using algorithmic differentiation to estimate the output error due to lowering the precision of variables. It provides accurate output error estimates while scaling better than previous mixed-precision efforts and thereby being more applicable to HPC workloads.
- A greedy algorithm for lowering the precision of variables and operations to achieve the specified accuracy.
- A floating-point precision sensitivity profile construction tool using ADAPT. This tool can be used for an application to guide programmers in the development of a mixed-precision version.
- Application of ADAPT to compare the sensitivities of algorithms to floating-point precision errors and identify the critical regions.
- An evaluation of our approach on different benchmarks and mini-applications achieving a 1.2x on LULESH, a proxy HPC application.

II. Overview

Consider the example of numerical anomalies introduced in [16] and [6] of using Simpson's rule to evaluate the integral of an input function f on a given interval (a, b) (for a given n, h is defined as $h = \frac{b-a}{2n}$):

$$\int_a^b f(x)dx = \frac{h}{3}[f(a) + 4f(a+h) + 2f(a+2h) + 4f(a+3h) + \cdots + 4f(a+(2n-1)h) + f(b)].$$

The corresponding C program to evaluate $\int_0^1 sin(\frac{\pi x}{2})dx = \frac{2}{\pi}$ for $n = 1000000$ is shown in Figure 1. This implementation uses the non-IEEE 80-bit `long double` precision provided by the C language (usually implemented using the x87 instruction set) and produces the answer 1.999999999999959. This program is subject to numerical anomalies due to cumulative rounding error where a large numbers of small values are added. When using IEEE `double` precision instead of `long double` precision, the answer is 2.000000000067576 with an error of $6.7e-11$. As demonstrated in [6], a mixed-precision version could use `long double` precision only for the two variables `s1` and `x`, with the rest of them stored in `double` precision to achieve a 1.6x speedup over the original version while maintaining the same final accuracy.

Another interesting point is that the rounding errors, which are due to adding two numbers with different numerical scales, start to accumulate in the later part of the iterations when the value of `s1` and `x` increase. This situation gives us another

```
long double fun(long double x) {
  long double pix = pi * x;
  long double result = sin(pix);
  return result;
}

int main( int argc, char **argv) {
  int l, i;
  const int n = 1000000;
  long double a = 0.0, b = 1.0, s1 = 0.0;
  long double h, x, tmp;
  h = (b - a) / (2.0 * n);
  x = a;
  s1 = fun(a);
  for(l = 0; l < n; l++) {
    x = x + h;
    s1 = s1 + 4.0 * fun(x);
    x = x + h;
    s1 = s1 + 2.0 * fun(x);
  }
  s1 = s1 + fun(b);
  tmp = h * pi / 3;
  s1 = s1 * tmp;
  printf("ans: %.15Le\n", s1);
  return 0;
}
```

Figure 1: **Computer program to evaluate integral of a function using Simpson's rule for numerical quadrature.** This program is subject to numerical anomalies. The tuned program with mixed precision uses `long double` only for the variables $s1$ and x and `double` for the rest to achieve the same final accuracy. There is another opportunity for optimization where $s1$ and x can be computed in lower precision in the initial iterations and then transitioning to higher precision in the middle of the execution.

opportunity to improve the performance by computing `s1` and `x` in lower precision in the beginning iterations and then transitioning the variables to higher precision in the middle of the execution.

We present a dynamic approach, implemented as a tool called ADAPT, that uses Algorithmic Differentiation (AD), an approach for computing the derivative of computer programs, to analyze floating-point precision sensitivity of variables and operations in a program. AD views a computer program as a composition of a sequence of arithmetic operations, and we use this to capture the propagation of errors through the data flow graph of the computation. ADAPT performs aggregation and analysis on this data along with the original computation to determine the floating-point sensitivity of all the variables and operations in the program. ADAPT also provides mixed-precision recommendations that satisfy a specified error threshold without requiring any search-based strategies.

III. Background

A. Floating Point Representation

IEEE floating-point arithmetic as standardized in 1985 (and subsequently revised in 2008 [17]) has become the primary implementation of real-valued arithmetic on digital computing platforms. Although it has well-known weaknesses (e.g., rounding error and cancellation), it is widely used in scientific computing because it provides a wide dynamic range and can be implemented efficiently in hardware.

Within a fixed-sized bit field, there are three parts: 1) the sign bit, 2) a biased exponent, and 3) a fractional part (sometimes called the *significand* or *mantissa*) with an implicit leading digit. The value stored is $(-1)^{sign} \cdot 1.frac \cdot 2^{exp}$. Using more bits provides more precision, but increases storage costs and computation time.

Rounding error occurs when a real number with infinite possibilities is approximated with a finite number of bits in a floating-point format. The relative rounding error of a value x in a specific precision is bounded by $\frac{1}{2^p}$ where p is the number of mantissa bits, because the error is bounded by the value represented by the least mantissa bit. The absolute error is bound by $|x| \times 2^{-p}$.

B. Algorithmic Differentiation

Algorithmic differentiation (AD), also known as "automatic" differentiation, is a chain-rule based technique to evaluate numerically the derivative of a function specified as a computer program [15]. Alternative methods for computing the derivatives include 1) hand-coding the derivatives, 2) symbolic differentiation, and 3) numerical differentiation. Although hand-coding derivatives can be more efficient, they tend to be tedious and error prone. Symbolic differentiation, available through systems such as Maple [18], requires the computer program to be represented as a single expression, which is difficult or impossible for reasonable-sized programs, especially in the presence of loops and complex data structures. Numerical differentiation using finite differences suffers from numerical instability due to rounding errors in discretization process.

Algorithmic differentiation considers a computer program as a sequence of elementary arithmetic operations and functions, applying the chain rule of differentiation to compute the derivatives. For a given function $y = f(u)$, where $u = g(x)$, application of chain rule gives us: $\frac{dy}{dx} = \frac{dy}{du} \times \frac{du}{dx}$.

AD has been applied to sensitivity analysis of a simulation to its input parameters as well as to optimization problems in the fields of fluid dynamics, engineering design, and climate modeling.

There are two modes of operation for AD: the forward mode and the reverse mode. The forward mode is also known as the "tangent linear" mode, and the reverse mode in known as the "adjoint" mode. In the forward mode, derivatives are computed using the chain rule, tracking the original computation. In the reverse mode, there is a forward sweep to collect information about the computation and a reverse sweep where chain rule is applied in the reverse order starting from the output. Forward-mode is preferred when there are larger number of outputs compared to inputs whereas reverse mode is preferred when there are large numbers of inputs. In the reverse mode, computation of the derivatives can be done at a small constant multiple of the original computation irrespective of the number of inputs [19].

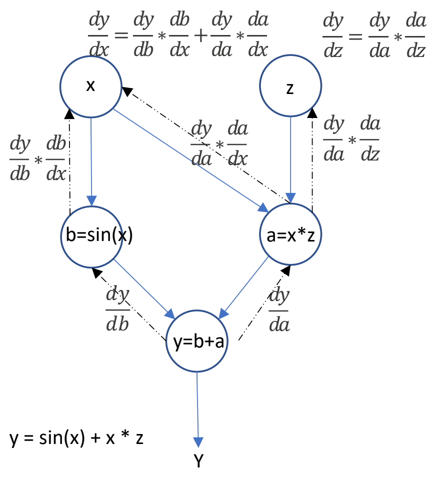

$\frac{dy}{dx} = \frac{dy}{db} * \frac{db}{dx} + \frac{dy}{da} * \frac{da}{dx}$
$\frac{dy}{dz} = \frac{dy}{da} * \frac{da}{dz}$
x
z
$\frac{dy}{db} * \frac{db}{dx}$
$\frac{dy}{da} * \frac{da}{dx}$
$\frac{dy}{da} * \frac{da}{dz}$
b=sin(x)
a=x*z
$\frac{dy}{db}$
$\frac{dy}{da}$
y=b+a
y = sin(x) + x * z
Y

Figure 2: **An example of the reverse mode of automatic differentiation.** The elementary operations are shown in circles and the edges represent data flow path. The dashed edges represent the computation of the derivatives, also referred to as *adjoint*, in the reverse mode. We can see that it computes the partial derivative of the output y with respect to all the inputs along the computation path.

Algorithmic differentiation is automated using tools that work by either code transformation or operator overloading. Source code transformation tools such as Tapenade [20], ADIC [21], ADIFOR [22] and OpenAD [23] take the source code and carry out a series of transformations to produce a final program that generates the derivatives. Other tools such as CoDiPack [24] and Adept [25] use operator overloading (often with expression templates for efficiency) to embed the derivative calculations into the given program. CoDiPack is designed for HPC applications and has support for MPI communication. We use CoDiPack for C++ programs and and Tapenade for C and Fortran programs, and we use the reverse mode of AD because we consider all variable values to be inputs so there is a high number of inputs compared to outputs.

IV. Mixed Precision Tuning

The main goal of ADAPT is to provide mixed-precision analysis by accurately estimating the output error due to changes in the precision of program variables. We propose an output error estimation model that is used to construct a mixed-precision version of the program if possible. In this section we describe how ADAPT estimates the output error using an error model based on algorithmic differentiation. We then go into the details of how ADAPT constructs the mixed-precision configuration that satisfies a specified error threshold. Finally, we describe the implementation details for the tool.

A. Output Error Estimation Model

We view the program as a function of inputs and various intermediate variables that uses compositions of elementary operations to compute some output. We are interested in the changes in the output as a result of small changes in various variables due to rounding errors. We construct our error model by using a first-order Taylor series approximation of the program, where the error in output is assumed to be linear in the rounding error. Let x be the input or the intermediate variable in the program and y be the target output variable given by $y = f(x)$, which signifies that y is a function of x and represents the part of the program that performs this operation. The first-order Taylor series approximation of the function $f(x)$ at $x = a$ is

$$\begin{aligned} y &= f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \dots, \\ &\approx f(a) + f'(a)(x-a). \end{aligned} \tag{1}$$

We use Equation 1 to estimate the error in the output, Δy, due to an error Δx in x:

$$\begin{aligned} \Delta y &= |f(a+\Delta x) - f(a)|, \\ &\approx |f(a) + f'(a)(a+\Delta x - a) - f(a)|, \\ &= |f'(a)(\Delta x)|. \end{aligned} \tag{2}$$

This can be extended to consider the output y as a function of several inputs,

$$y = f(x_1, x_2, \dots, x_n),$$

With $\overline{x} = \{x_i\}$ as a vector of multiple inputs and generalizing Equation 2, we obtain the following estimated output error Δy due to error in the inputs at $\overline{x} = \overline{a}$, where $x_i = a_i,\ i \in \{1, \dots, n\}$:

$$\Delta y \approx |f'_{x_1}(\overline{a})\Delta x_1| + \dots + |f'_{x_n}(\overline{a})\Delta x_n|, \tag{3}$$

$$= \sum_i |f'_{x_i}(\overline{a})\Delta x_i|, \tag{4}$$

$$= \overline{|f'(a)^T \overline{\Delta x}|}. \tag{5}$$

Here $f'_{x_i}(\overline{a})$ represents the partial derivative with respect to x_i evaluated at $\overline{x} = \overline{a}$ and the bars represent the vector form. Equation 5 is a linear approximation that uses only first order partial derivatives (one for each variable) as computed via the chain rule on the dependency graph. As a result, in the case of simultaneous errors in multiple variables we ignore the dependencies between these variables, treating them as independent variables, and compute the total change as a sum of individual changes. Our approach relies on the assumption that the resultant change in any variable is small enough such that the corresponding partial derivative is still valid. This is a reasonable assumption in our case because we expect any change of precision to result in small errors.

For a specific precision, the absolute error due to rounding in variable x is less than $|x|\epsilon$, where $\epsilon = 2^{-p}$ and p is the number of bits in the mantissa for that precision. We use this as a measure of error and define a metric $\mathcal{E}$ to capture the sensitivity of any input or intermediate variable to rounding errors:

$$\mathcal{E}_x = |f'(x) \times x|. \tag{6}$$

The error metric in Equation 6 captures both the sensitivity of the variables based on the adjoint information as well as the impact of rounding errors. In addition, it provides a relative measure of sensitivity between the variables in a program; a variable with a larger error metric is more sensitive to roundoff and has a larger contribution to the final output error. We use this metric to identify instability in programs, to compare different algorithms, and to construct a floating-point precision sensitivity profile.

B. Mixed Precision Allocation

To identify a set of variables that can be changed to lower precision without violating a given error threshold, we begin by evaluating the output error due to lowering the precision at every dynamic instance of each variable. We then aggregate the output error estimates at the assignment statement level as well as at the source variable and function level.

For every dynamic instance of variable x (denoted x_i), we compute the output error contribution Δy_{x_i} resulting from conversion to lower precision as

$$\begin{aligned} \Delta y_{x_i} &= f'(x_i)\Delta x_i, \\ &= f'(x_i)(x_i - x_i^{lower}). \end{aligned} \tag{7}$$

Here, Δx_i is the estimated error introduced in x_i due to lowering the precision. The aggregated error over all the dynamic instances of the variable is then defined as follows:

$$\Delta y_x = \sum_{i=1}^{n} \Delta y_{x_i}. \tag{8}$$

Using Equation 5 we can say that the output error, as a result of change in precision of multiple variables, can be computed by taking a summation over their individual contribution given by Equation 8. We use this as the basis of our mixed-precision allocation and describe a greedy approach used to create mixed-precision configuration (see Algorithm 1).

First, the algorithm iterates over all the dynamic instances of variables, *vars*, and aggregates error over all the dynamic instances of variable *var* in *OutputError(var)*. Next, the variables are sorted in increasing order of their contribution to the final output given by Equation 8. Finally, variables are assigned to lower precision if converting them does not violate the error threshold.

This algorithm converts as many variables as possible to lower precision. We provide two variations of this greedy approach in ADAPT. The first one lowers the precision of variables in increasing order of error per dynamic count. That means, in the case of two variables with the same output error, this approach gives priority to the variable that has more dynamic instance (an indication of it being more expensive). The second variation iterates over the functions in increasing order of the output error contribution and lowers the precision of variables that don't violate the error threshold. This grouping can sometimes be beneficial in reducing the number of implicit as well as explicit cast operations.

```
input  : A list of variables $vars$
input  : $TolError$
output: $S$ containing variables in lower precision
begin
    for $var \in vars$ do
        $InputError \leftarrow var.value - var.value\_lower$
        $OutputError(var) \leftarrow OutputError(var) +$
          $var.adjoint \times InputError$
    end
    sort $OutputError$ in increasing order
    $var \leftarrow RemoveMinOutputError()$
    while $var \neq null$ and
     $TolError - |OutputError(var)| \geq 0$ do
        $TolError \leftarrow TolError - |OutputError(var)|$
        $S \leftarrow S + \{var\}$
        $var \leftarrow RemoveMinOutputError()$
    end
end
```
Algorithm 1: Greedy mixed-precision allocation algorithm

```
double fn(double Sn, int n) {
  double e = Sn*Sn;
  double tmp = sqrt(4 - e);
  double Sn1 = sqrt(2-tmp);
  return Sn1;
}
```
(a) Flagged source of instability in algorithm 1

```
double fn(double Tn, int n) {
  double e = Tn * Tn;
  double tmp = sqrt(4 + e);
  double Tn1 = 2* Tn / (2+tmp);
  return Tn1;
}
```
(b) No error flagged in algorithm 2

Figure 3: **Illustration of two algorithms for approximating π.** Here we compare the sensitivity of two algorithms to rounding errors. ADAPT flags the source of instability for algorithm 1. Algorithm 2, which has been rewritten to avoid inaccuracy, is not flagged.

C. Implementation Details

There are two stages in ADAPT: 1) evaluation of adjoints using an AD tool and 2) mixed-precision allocation.

Traditionally, AD tools have been applied in sensitivity analysis, where the adjoints are computed for the inputs with respect to the output. Here, we consider all the variables and their intermediate results as potential points for optimization. Therefore, we need to obtain the adjoints at every location we want to analyze, which is typically at every assignment operator. We use an AD tool (CoDiPack or Tapenade) to compute the partial derivative $\frac{\partial y}{\partial x_i}$ for all the input and intermediate variables x_i with respect to an output y. CoDiPack provides support for C++ programs, while Tapenade supports C and Fortran codes. ADAPT interfaces with the user program and these AD tools by providing APIs to register the variables and assignment statements of interest. Regardless of the backend used, we modify the original program's source to label the variables of interest using a simple ADAPT API.

CoDiPack: When using CoDiPack as the AD backend, ADAPT registers the variables of interest with the CoDiPack tool at runtime. Because CoDiPack employs operator overloading and template expressions to obtain the partial derivatives, the program must also be converted to use `AD_real` instead of floating point types. We provide macros to assist with this conversion.

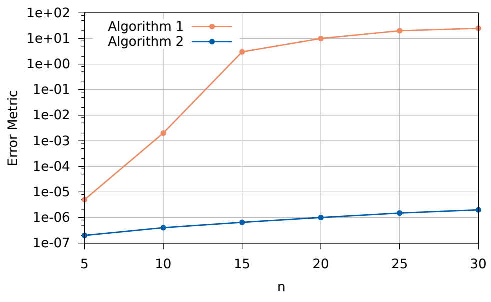
Tapenade: When using Tapenade as the AD backend, the ADAPT API calls are processed during the source-to-source transformation to produce code that calculates the partial derivatives. The transformed program is then executed with a representative input to obtain adjoints of all dynamic instance of the input as well as the intermediate variables in the program.

For both tools, an API callback retrieves the adjoint (the partial derivative) as well as the value of the variable, which is then used in the next step for mixed-precision allocation described in section IV-B.

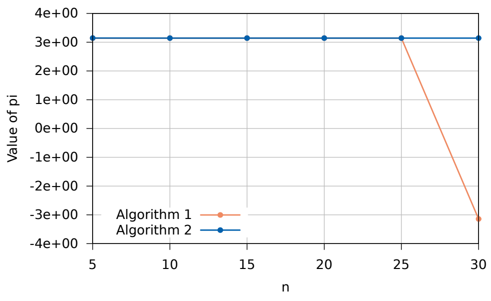
Usage: We run the ADAPT tool for a given program on a given input with a specified error threshold. ADAPT generates a report of floating-point precision sensitivity for every static assignment statement using the error metric defined in Equation 6. It also generates a mixed-precision configuration using the algorithm described in section IV-B. This configuration is used as a guide by the developer to create mixed-precision version of the program.

V. Evaluation

We evaluate ADAPT on six benchmarks and one proxy application. First we show the application of our approach on the Archimedes despair problem, where we compare the sensitivities of two algorithms to floating-point precision errors and identify problematic regions of the code. Next, we evaluate the capability of ADAPT in identifying variables that can be in lower precision on four different benchmarks for different error thresholds. We also use our approach to construct a floating point precision sensitivity profile for HPCCG, a mini-app in the Mantevo benchmark suite, as well as LULESH, a proxy HPC application. We use these profiles to develop mixed-precision versions of these benchmarks and to evaluate performance gains. We ran all CPU experiments on Quartz, a cluster containing Intel Xeon E5-2695 processors with 2.1 GHz cores and 128 GB of memory per node. We also use Blue Waters supercomputer, a Cray machine at NCSA, for the evaluation of LULESH. We use the XK7 nodes on Blue Waters that include NVIDIA GPUs. We ran each experiment five times and report the average time.



(a) Error metric comparison.



(b) Computed π value comparison.

Figure 4: **ADAPT analysis on Archimedes problem.** Results indicate a rapid growth in the error metric with increasing n for algorithm 1 compared to algorithm 2, suggesting instability in algorithm 1. While algorithm 2 continues to converge to π, algorithm 1 computes wrong result after 30 steps.

A. *Archimedes Despair Problem*

In order to demonstrate the capability of ADAPT to identify stability issues in a numerical algorithm, we consider the Archimedes despair problem [26]. The problem describes a method for approximating π based on the length of the perimeter of a polygon inscribed in a circle. This method follows the following recurrence: $S_{n+1} = 2^n\sqrt{2(1-\sqrt{1-\epsilon^2})}$ where $\epsilon = S_n/2^n$. For large n, $S_n \to \pi$. However, ϵ becomes very small and unrepresentable by the choice of floating point precision. The result is an inaccurate evaluation of the inner square root term $(1-\sqrt{1-\epsilon^2}) = \gamma$, leading to an unstable recurrence. Rewriting γ by rationalizing yields a recurrence formula that is more accurately evaluated and avoids this instability. Simplifying terms and following [26], this recurrence takes the form: $T_{n+1} = \frac{T_n}{2(1+\sqrt{1-T_n})}$ where $Tn = (\frac{S_n}{2})^2$.

Figure 3 gives an illustration of both algorithms. Using the adjoint information and the corresponding computed absolute error (see Equation 2) in the variables, the tool correctly flags the intermediate variable `tmp` in line 6 of algorithm 1 as the source of instability. This is further highlighted in Figure 4a, where the error metric, $\mathcal{E}$, evaluated on the two algorithms rapidly grows for algorithm 1 compared to algorithm 2. Figure 4b illustrates the behavior of the different algorithms as $S_n \to \pi$. Here, we observe that the more stable algorithm 2 progresses in its approximation of π, whereas algorithm 1 drops to zero after about 30 steps.

B. *Linear Solver with Preconditioner*

In this section, we demonstrate the use of our tool to evaluate the behavior of an iterative solver for solving a linear system of equations. We consider a reservoir modeling application characterized by the linear system matrix PORES3 from the University of Florida Sparse Matrix collection [27]. The matrix is real, nonsymmetric, and has a condition number estimate of roughly 1.0e-05. The right-hand-side of the linear system is generated artificially by assuming a solution of all ones. The iterative method we consider is the generalized minimal residual method (GMRES) [28]. The algorithm uses the Arnoldi iteration to compute the minimal residual Krylov subspace vector that approximates the solution to the linear system. In this example, we select the quantity of interest to be the solution error. Therefore, we assume convergence when the error in the solution is reduced below 1.0e-10. We evaluate the behavior of the iterative solver without preconditioning and with (correct) preconditioning. For the case with preconditioning, we use the threshold-based incomplete LU factorization preconditioner (ILUT) [29]. The fill-factor (or memory usage) of the resulting preconditioner is approximately 1.5.

Evaluating the linear solver with ADAPT, we identified regions of the algorithm with the largest error metric. The results are presented in Table I. Here, "Res. Norm" refers to the evaluation of the residual norm during each iteration of the GMRES algorithm; "Modified GS" refers to the modified Gram-Schmidt process within the Arnoldi step to compute the Hessenberg matrix; "Hessenberg update" refers to the transformation of the Hessenberg matrix into upper triangular form by the use of plane rotations; and "Triangular solve" refers to the solution of the upper triangular system to compute the solution of the linear system.

The results indicate that the GMRES algorithm favors the preconditioned linear system compared to the ill-conditioned system, which is well-known in the literature. However, ADAPT is able to identify key components of the algorithm that are sensitive to ill-conditioning. Furthermore, we note that, for this linear system, the preconditioner construction (ILU factorization) and the preconditioner solve operations produced an error metric on the order of 1.0e-15. This result suggests that, while the preconditioner is important to improve the behavior of the iterative solver, the preconditioning operations are less sensitive to numerical accuracy issues and are quite stable for this problem. We note that this result pertains only to this linear system and may be inapplicable to other linear systems. However, this result aligns with the literature in the context of mixed-precision linear solvers, where the preconditioning operations are done in lower precision. For example, half-precision arithmetic is used in [30] to accelerate dense linear solvers.

C. *Benchmarks*

We evaluate ADAPT on four benchmarks: 1) arclength, 2) simpsons, 3) jetEngine and, 4) carbonGas. arclength and

Max. Error Metric	No Preconditioner	With Preconditioner
Res. Norm	3.56e-02	8.15e-05
Modified GS	7.14e+01	1.43e-03
Hessenberg update	2.41e-02	1.92e-05
Triangular Solve	1.57e+00	7.13e-04
Number of iters	1000	100

Table I: **Error metric for different code regions of an iterative solver.** Lower value of metric corresponds to higher stability. Our prediction of higher stability for the preconditioned system agrees with the well-known result in the literature that GMRES algorithm favors the preconditioned linear system.

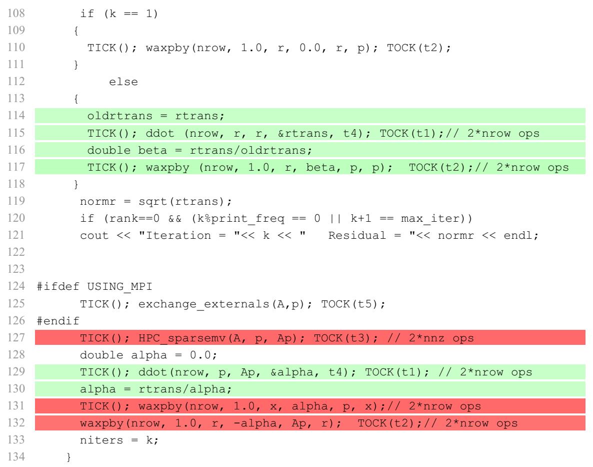
Program	# vars	Error Threshold	# vars in lower prec	Actual Error	Estimated Error
arclength	9	1e-12	9	1.7e-13	1.7e-13
simpsons	10	1e-12	9	4.5e-14	3.7e-14
jetEngine	28	1e-13	14	2.5e-13	8.4e-14
carbonGas	15	1e-10	7	1.0e-11	2.8e-11

Table II: **Output error estimates and mixed-precision results for benchmarks using ADAPT.** Here we show the mixed-precision configuration as well as the output error estimates given by ADAPT for different error threshold values. ADAPT was able to accurately estimate the final output error.

simpsons were used in the evaluation of Precimonious [6], a search-based method for mixed-precision tuning. jetEngine and carbonGas were used in the evaluation of FPTuner [31]. Table II lists the results of our case studies. The initial configuration for all the benchmarks is in `long double`, consistent with their original publication. We choose an error threshold for each benchmark, where the error threshold specifies the absolute error difference with respect to the initial higher-precision version of the program. Table II shows the estimated as well as actual errors for mixed-precision configuration suggested by ADAPT.

Table II shows that our approach is able to estimate the final output error accurately and to identify mixed-precision configurations within the specified error threshold for most of the benchmarks. In the case of jetEngine, ADAPT slightly underestimates, and therefore goes above the set threshold. The analysis time is 17 seconds for arclength, 7 seconds for simpsons and on the order of few microseconds for jetEngine and carbonGas benchmarks.

In addition to the mixed-precision configuration of original program variables, ADAPT provides the precision sensitivity of all the intermediate variables generated dynamically throughout execution. This additional information enabled us to explore alternative mixed-precision for the simpsons benchmark, where the variables were assigned `double` precision in the initial 10000 iterations and then converted to higher precision (`long double`). Table III shows the speedup obtained for these benchmarks at an error threshold of 1e-12. Even with this tight error threshold, we are able to identify configurations that yield a speedup of at least 1.1 for simpsons and arclength.



```
      if (k == 1)
     {
       TICK(); waxpby(nrow, 1.0, r, 0.0, r, p); TOCK(t2);
     }
          else
     {
       oldrtrans = rtrans;
       TICK(); ddot (nrow, r, r, &rtrans, t4); TOCK(t1);// 2*nrow ops
       double beta = rtrans/oldrtrans;
       TICK(); waxpby (nrow, 1.0, r, beta, p, p);  TOCK(t2);// 2*nrow ops
     }
      normr = sqrt(rtrans);
      if (rank==0 && (k%print_freq == 0 || k+1 == max_iter))
      cout << "Iteration = "<< k << "   Residual = "<< normr << endl;


#ifdef USING_MPI
      TICK(); exchange_externals(A,p); TOCK(t5);
#endif
      TICK(); HPC_sparsemv(A, p, Ap); TOCK(t3); // 2*nnz ops
      double alpha = 0.0;
      TICK(); ddot(nrow, p, Ap, &alpha, t4); TOCK(t1); // 2*nrow ops
      alpha = rtrans/alpha;
      TICK(); waxpby(nrow, 1.0, x, alpha, p, x);// 2*nrow ops
      waxpby(nrow, 1.0, r, -alpha, Ap, r);  TOCK(t2);// 2*nrow ops
      niters = k;
    }
```

Figure 5: **Output of ADAPT analysis highlighting lines of code of HPCCG benchmark and their precision requirement.** Regions that are highlighted in red require higher precision and those in green can be in lower precision. The critical variables associated with the higher precision sections are solution x, residual r, and the matrix-vector product Ap.

D. HPCCG

We evaluate our approach on HPCCG, a mini-application from the Mantevo benchmark suite. It is a conjugate gradient benchmark code for a 3D chimney domain with a problem size of $20 \times 30 \times 160$. Figure 5 shows a code highlight of the critical sections of the algorithm that need to be evaluated in higher precision as determined by ADAPT. The critical variables associated with these sections are identified as the solution x, the residual r, and the variable Ap, which is the result of the matrix-vector product between the matrix A and the search direction p. At each iteration of the algorithm, r and Ap are used to compute the step-length to update the solution x. However, perhaps more importantly, r and Ap are used to ensure that the subsequent search directions are conjugate to each other. As a result, inaccuracies in their calculation can lead to an unstable conjugate gradient algorithm.

Further analyses of the results reveal that initial iterates are critical for accuracy. Figure 6 shows a plot of the critical variables across different iterations. The results indicate that after the first 20 iterations, the error in r and Ap are below the 1.0e-10. Beyond 60 iterations, the error in the solution is also below this threshold. These results are interesting as they suggest the feasibility of performing lower precision arithmetic in the later iterations for efficiency.

E. LULESH

LULESH is a proxy application developed at LLNL. It approximates hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh. We use ADAPT on LULESH 2.0 to create a precision sensitivity profile of the program. Figure 7 shows the call graph at the function level highlighting functions that need to be in higher precision. The profile generated

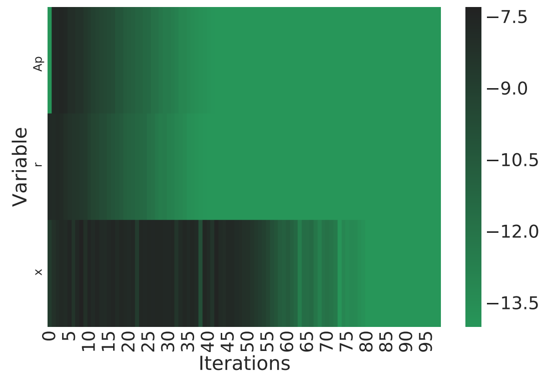


Figure 6: **Output error estimate given by ADAPT across iterations for HPCCG.** Result from the ADAPT analysis reveals that initial iterates are critical (depicted in darker shade) for accuracy. Beyond 60 iterations, the error in the solution x is below the threshold of wer precision arithmetic in the later iterations.

by ADAPT provides details at every variable assignment but, because it is not feasible to show the entire code, we show this aggregated metric at the function level.

According to our analysis, the function *CalcTimeConstraintsForElems*, which pertains to calculation of the time step contraints, can be performed in lower precision. This result is in accordance with our expectation, because the time step variable is merely used to advance the simulation and therefore needs to be accurate only to a few significant digits. The routine *CalcElemCharacteristicLength* computes the characteristic length of the element, which is later used in *CalcTimeConstraintsForElems* to update the next time step. As a result, it is not surprising that this operation can also be performed in lower precision.

CalcElemShapeFunctionDerivatives computes the determinant of the element Jacobian matrix, which is then used in *CalcElemVelocityGradient* to compute the rate of distortion of the element volume. This distortion variable is used as a conditional to switch on artificial viscosity and is not used directly in the computation of other intermediate variables in the code. Note that, because conditional statements are non-differentiable, the AD approach cannot evaluate the impact of conditional variables. We discuss this further in Section VI.

We used the profile as a guide to develop a mixed-precision version for a CUDA implementation of LULESH. The mixed-precision version was within the error threshold specified in [32], which resulted in a speedup of 1.2x.

F. Comparison with Existing Tools

In this section, we compare ADAPT with Precimonious [6] and FPTuner [31]. Precimonious is a search based tool, and FPTuner [31] is a rigorous error analysis method based on symbolic taylor expansion and interval functions. Note that CRAFT [5] is another search based tool that we evaluated. However, CRAFT is not directly comparable with ADAPT because ADAPT analyzes variables while CRAFT analyzes instructions. Moreover, CRAFT searches took significantly longer (e.g., over two minutes for arclength and over three minutes for simpsons, even with 36-way parallelism).

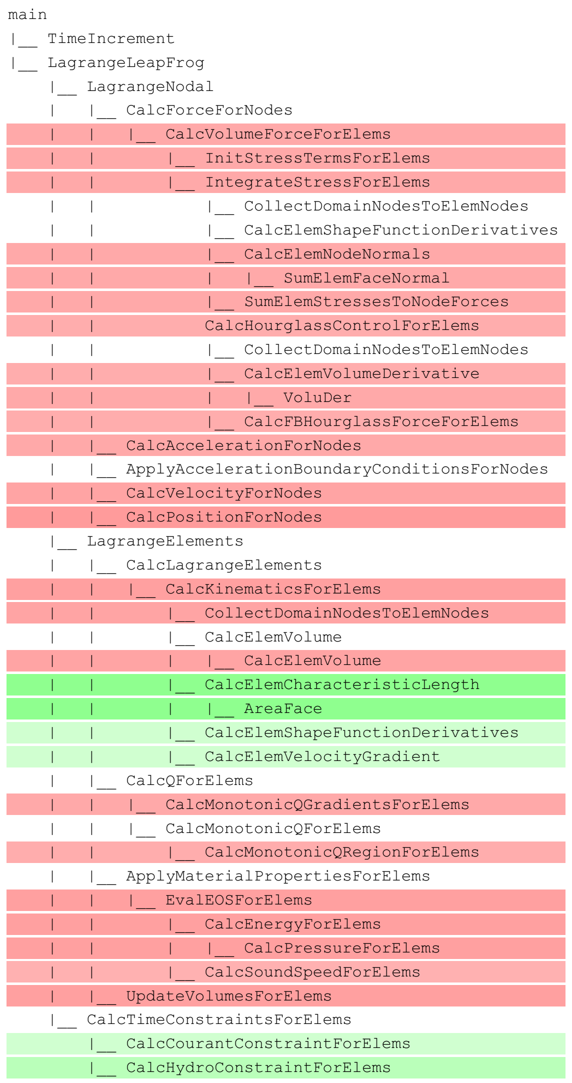

```
main
|__ TimeIncrement
|__ LagrangeLeapFrog
    |__ LagrangeNodal
    |   |__ CalcForceForNodes
    |   |   |__ CalcVolumeForceForElems
    |   |       |__ InitStressTermsForElems
    |   |       |__ IntegrateStressForElems
    |   |           |__ CollectDomainNodesToElemNodes
    |   |           |__ CalcElemShapeFunctionDerivatives
    |   |           |__ CalcElemNodeNormals
    |   |           |   |__ SumElemFaceNormal
    |   |           |__ SumElemStressesToNodeForces
    |   |           CalcHourglassControlForElems
    |   |           |__ CollectDomainNodesToElemNodes
    |   |           |__ CalcElemVolumeDerivative
    |   |           |   |__ VoluDer
    |   |           |__ CalcFBHourglassForceForElems
    |   |__ CalcAccelerationForNodes
    |   |__ ApplyAccelerationBoundaryConditionsForNodes
    |   |__ CalcVelocityForNodes
    |   |__ CalcPositionForNodes
    |__ LagrangeElements
    |   |__ CalcLagrangeElements
    |   |   |__ CalcKinematicsForElems
    |   |       |__ CollectDomainNodesToElemNodes
    |   |       |__ CalcElemVolume
    |   |       |   |__ CalcElemVolume
    |   |       |__ CalcElemCharacteristicLength
    |   |       |   |__ AreaFace
    |   |       |__ CalcElemShapeFunctionDerivatives
    |   |       |__ CalcElemVelocityGradient
    |   |__ CalcQForElems
    |   |   |__ CalcMonotonicQGradientsForElems
    |   |   |__ CalcMonotonicQForElems
    |   |       |__ CalcMonotonicQRegionForElems
    |   |__ ApplyMaterialPropertiesForElems
    |   |   |__ EvalEOSForElems
    |   |       |__ CalcEnergyForElems
    |   |       |   |__ CalcPressureForElems
    |   |       |__ CalcSoundSpeedForElems
    |   |__ UpdateVolumesForElems
    |__ CalcTimeConstraintsForElems
        |__ CalcCourantConstraintForElems
        |__ CalcHydroConstraintForElems
```

Figure 7: **Output of analysis from ADAPT highlighting precision requirements of different functions in LULESH.** Functions in LULESH highlighted in red require higher precision whereas the ones in green can be in lower precision. ADAPT analysis indicates that the function *CalcTimeConstraintsForElems*, which pertains to the calculation of time step constraints, can be performed in lower precision. Functions *CalcElemShapeFunctionDerivatives* and *CalcElemVelocityGradient*, which compute the rate of distortion of the element volume, can also be evaluated in lower precision.

Table IV shows the overhead of Precimonious, FPTuner and ADAPT for some of the benchmarks. FPTuner is applicable to benchmarks written in real-valued expression language. There is no real-valued expression language version of LULESH and HPCCG and it is not trivial to convert them to that form. Since ADAPT supports only C and C++, we use C version of the

Program	Error Threshold	Output Error	Speedup
HPCCG	1e-10	2.8e-15	1.10
arclength	1e-12	1.7e-13	1.11
simpsons	1e-12	4.5e-14	1.13
jetEngine	1e-13	2.5e-13	1.40
carbonGas	1e-10	1.0e-11	1.57
Lulesh (GPU)	1e-08	1.8e-11	1.20

Table III: **Performance speedup using mixed-precision configuration suggested by ADAPT.** Analysis from the ADAPT was used to create mixed-precision version of the program which shows speedup for many of the programs evaluated. LULESH benefits most from the mixed-precision optimization with a speedup of 1.2x.

Program	Error Threshold	App Time (s)	Precimonious (s)	FPTuner (s)	ADAPT (s)
HPCCG	1e-10	3.3e-1	3.6e+2	-	3.7e+1
arclength	1e-12	2.0e-1	1.1e+2	-	1.7e+1
simpsons	1e-12	5.2e-2	4.2e+1	-	8.5e+0
jetEngine	1e-13	7.0e-8	5.2e+1	7.3e+1	1.2e-6
carbonGas	1e-10	3.3e-7	2.7e+1	2.0e+1	2.0e-5

Table IV: **Comparison of Precimonious, FPTuner and ADAPT.** This shows the analysis time for Precimonious, FPTuner and ADAPT as well as the application time. ADAPT is orders of magnitude faster than Precimonious and FPTuner.

programs (carbonGas and jetEngine) generated by FPTuner. Precimonious doesn't work well with C++ dynamic memory allocation, causing it to crash. Therefore, we are unable to get the comparison results for LULESH. Precimonious doesn't have support for changing the type of dynamic memory allocation variables. Therefore, in the case of HPCCG, Precimonious search didn't explore 50% of the variables in the program.

Table IV shows the comparison of Precimonious with ADAPT for a specified threshold. Precimonious explored 800 configurations for HPCCG, around 100 configurations for arclength, simpsons, and jetEngine, and 60 for carbonGas. The analysis time of Precimonious and FPTuner was orders of magnitude more than that of ADAPT. In the case of jetEngine and carbonGas, high analysis time of Precimonious is attributed to high overhead of executing LLVM pass to transform the LLVM bitcode based on the search configuration. For arclength and simpsons, Precimonious and ADAPT selected the same set of variables to convert to lower precision to maintain the specified error threshold. However, for HPCCG the mixed-precision versions were different. The mixed precision version generated using analysis from ADAPT factored in precision requirement across iterations and used lower precision as iterations progressed to give 1.1x speedup. This level of fine-grained information is not available with Precimonious and therefore the configuration suggested by it did not result in any speedup.

VI. Discussion

In this section, we summarize the capabilities of ADAPT for mixed-precision analysis, discuss the current limitations of our work, and suggest possible directions for future work.

A. *Summary of* ADAPT*'s Capabilities*

ADAPT is able to highlight regions of code that are highly-sensitive to floating-point precision as well as to identify variables and operations that can be converted to lower precision. This profile informed algorithmic choices in the Archimedes despair program by capturing the stability issues of particular operations. The error output estimates of the mixed-precision configuration suggested by ADAPT closely match the real output error and is well within the desired error threshold. ADAPT provides a very detailed view of precision sensitivity at every assignment statement as well as across multiple iterations, enabling us to create mixed-precision configurations that take advantage of inter-iteration behavior. Finally, ADAPT is able to provide insights codes in a scalable fashion, making it feasible to apply it to LULESH, an HPC proxy application, to achieve a 1.2x speedup.

B. *Current Limitations*

While ADAPT provides a practical approach with accurate estimates of output error for mixed-precision tuning, there are still several limitations.

Analysis limited to the input used: Our approach does not provide guarantees for all possible inputs. ADAPT is a dynamic approach where the precision analysis is done on a particular dataset. Often, application developers have model datasets (e.g., benchmarks) that capture various characteristics of different possible inputs. We can use these representative datasets to provide a mixed-precision configuration for the program in general.

Control flow divergence: The branches in a program are not differentiable. Therefore, the analysis provided by ADAPT cannot factor in the potential alteration to the control flow path due to lowering of precision that can result in an erroneous output. To the best of our knowledge, this is still an open problem in AD and beyond the scope of this work. However, we conjecture that a shadow-value tool such as [33] could help identify instances of control flow divergence between two precisions.

Non-differentiable functions: Many functions are not differentiable at certain input values. For example, $1/x$ is not differentiable at $x = 0$. In such cases, we could compute the derivative using a small perturbation.

Memory requirements: The reverse mode of AD has the advantage that it can compute the adjoints in a constant multiple of the original computation for any number of inputs. To do this, it has to store the relevant intermediate computations (often referred to as a "tape"). Depending on the algorithm and the size of the input problem, storing the tape can be very expensive. Periodic checkpointing could help to mitigate this cost.

Overhead of type cast: Mixed precision assignment may result in implicit and explicit cast operations potentially leading to degradation of performance. Therefore, while creating the mixed-precision version of the program, we count the number of casts using Intel's XED2 library included in the Pin tool. We use a simple cost model similar to the cost function in [10] that takes into account the number of single and double precision operations as well as the number of casts in order to compare the performance of the transformed program.

VII. Related Work

There have been various efforts to create mixed-precision versions of an application manually both for CPU as well as GPU codes [3], [34], [35], [4], [36]. These works have shown the potential of mixed-precision for performance improvement and energy savings. Manually creating mixed-precision configurations for an application requires extensive knowledge of the numerical behavior of the algorithm and becomes challenging or infeasible for a large programs with multiple modules interacting with each other.

Floating-point error analysis has been studied extensively in literature. Benz et al. [13] presented a dynamic analysis approach for finding accuracy problems, where they store the higher precision computation in a shadow value and report a problem if it differs significantly from the original value. Dynamic analysis has also been used by Lam et al. [37] to detect cancellation errors. Bao et al. [11] and Nathan et al. [12] propose an approach that monitors the application to automatically discover unstable floating-point executions. Several static analysis based rewriting approaches have been proposed to minimize the rounding error [10], [8], [38], [9].

There have also been various static and sound approaches proposed for floating-point source code to provide rigorous bounds on the potential rounding error. Dinechin et al. [39] presented a tool called Gappa based on interval analysis. Magron et al. [40] proposed an optimization technique using semidefinite programming to provide formal rounding error bounds, implemented in their tool called Real2Float. Fluctuat [41] defines several abstract semantics to statically track the error of a floating-point program. Solovyev et al. [7] proposed FPTaylor, which models errors using symbolic Taylor expansion and uses global optimization for rigorous error bounds. FPTuner [31] uses a modified version of symbolic Taylor expansion to study rounding error under generic precision allocation. Darulova et al. [14] combined affine arithmetic with SMT-solving to tune the precision of real-valued expressions in Rosa and Daisy [42], [10]. These tools provide rigorous guarantees of accuracy, often giving very tight bounds on rounding error. However, they generally do not scale well and are targeted towards smaller programs and kernels that can be verified statically.

Algorithmic differentiation has been used to study the effects of rounding errors [43]. Langlois [44] proposed a new method called CENA which used AD to compute a correction term for elementary rounding errors to improve the accuracy. Gaffar et al. [45], [46] has used AD for bit-width optimization for FPGAs. Similarly, Vassiliadis et al. [47] proposed a method that combined interval analysis and algorithmic differentiation for significance analysis. None of these efforts targeted mixed-precision implementations directly, and neither effort attempted to apply their work to HPC workloads.

Other approaches use some form of search-based optimization for mixed-precision configurations. CRAFT [5], [48] detects precision requirements for various program components, providing guidance for building mixed-precision versions of a program. Precimonious [6] uses a delta-debugging algorithm to explore mixed-precision configurations. Precimonious was further extended in HiFPTuner [49] to use dependence analysis to improve the search algorithm. Graillat et al. [50] proposed a similar approach for tuning, but used discrete stochastic arithmetic (DSA). The main drawback of these approaches is that exploring even a subset of the state space is very time-intensive because the state space to explore is exponential in the number of variables. The search-based algorithms can also get trapped in local minima.

VIII. Conclusion

The primary goal of our work was to provide a scalable approach for mixed-precision analysis on HPC workloads while providing more rigorous guarantees about the selected mixed-precision configuration. We used algorithmic differentiation (AD) to estimate the error in the final output due to lowering the precision of variables. In contrast to the time-intensive search-based approaches used for identifying valid mixed-precision configurations, we used AD to provide floating-point precision sensitivity of programs at a constant multiple of the original computation, irrespective of the number of variables explored. The precision sensitivity profile, which highlights regions of the code that can potentially be converted to lower precision, was used as a guide to study the stability of the algorithm and to develop mixed precision configurations of the program.

We evaluated ADAPT on six benchmarks and a proxy application. We found that ADAPT is able to provide critical information about the floating-point sensitivity of the code. The profile was used to make algorithmic choices, as shown in the case of Archimedes despair problem. It was also used for identifying stability issues and was able to accurately capture the operation causing the instability. ADAPT was also used as a guide to build mixed-precision configurations to achieve performance improvement. The estimates of the output error for a mixed-precision configuration suggested by ADAPT closely matches the real output error well within the desired error threshold. Most importantly, ADAPT is able to provide insights about the floating-point sensitivity of codes in a scalable fashion, making it feasible to apply it to LULESH, an HPC proxy application, to achieve 1.2x speedup.

Acknowledgments

We thank Garrett Folks for his feedback on a draft of this paper. We thank the anonymous reviewers for their suggestions for improving this paper. In particular, we thank our shepherd for working with us to get our paper ready for acceptance. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, via LDRD project 17-SI-004. IM release number LLNL-CONF-748742.

References

[1] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, X. Chi, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Z. Jin, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. Mueller, W. E. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. van der Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick, "The International Exascale Software Project Roadmap," International Journal of High Performance Computing Applications, vol. 25, no. 1, pp. 3–60, 2011. [Online]. Available: http://hpc.sagepub.com/cgi/doi/10.1177/1094342010391989

[2] G. Gopalakrishnan, P. D. Hovland, C. Iancu, S. Krishnamoorthy, I. Laguna, R. A. Lethin, K. Sen, S. F. Siegel, and A. Solar-Lezama, "Report of the HPC Correctness Summit, Jan 25–26, 2017," Washington, DC, Tech. Rep., 2017. [Online]. Available: http://arxiv.org/abs/1705.07478

[3] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov, "Accelerating scientific computations with mixed precision algorithms," Computer Physics Communications, vol. 180, no. 12, pp. 2526–2533, 2009.

[4] R. Medhat, M. O. Lam, B. L. Rountree, B. Bonakdarpour, and S. Fischmeister, "Managing the Performance/Error Tradeoff of Floating-point Intensive Applications," in Proceedings of the International Conference on Embedded Software (EMSOFT'17). ACM, 2017.

[5] 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. ACM, 2013, pp. 369–378.

[6] 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 Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis on (SC'13). New York, New York, USA: ACM Press, nov 2013, pp. 1–12.

[7] A. Solovyev, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan, "Rigorous estimation of floating-point round-off errors with symbolic taylor expansions," in International Symposium on Formal Methods. Springer, 2015, pp. 532–550.

[8] N. Damouche, M. Martel, and A. Chapoutot, "Intra-procedural Optimization of the Numerical Accuracy of Programs," in Proceedings of the 20th International Workshop on Formal Methods for Industrial Critical Systems, FMICS 2015, vol. 9128, 2015, pp. 31–46. [Online]. Available: http://link.springer.com/10.1007/978-3-319-19458-5{_}3

[9] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2015, pp. 1–11, 2015. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2737924.2737959

[10] E. Darulova, E. Horn, and S. Sharma, "Sound mixed-precision optimization with rewriting," in Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems, ser. ICCPS '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 208–219. [Online]. Available: https://doi.org/10.1109/ICCPS.2018.00028

[11] T. Bao and X. Zhang, "On-the-fly detection of instability problems in floating-point program execution," in ACM SIGPLAN Notices, vol. 48, no. 10. ACM, 2013, pp. 817–832.

[12] R. Nathan, H. Naeimi, D. J. Sorin, and X. Sun, "Profile-driven automated mixed precision," arXiv preprint arXiv:1606.00251, 2016.

[13] 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.

[14] E. Darulova and V. Kuncak, "Sound compilation of reals," in Acm Sigplan Notices, vol. 49, no. 1. ACM, 2014, pp. 235–248.

[15] U. Naumann, The art of differentiating computer programs: an introduction to algorithmic differentiation. Siam, 2012, vol. 24.

[16] D. H. Bailey, "Resolving numerical anomalies in scientific computation," Tech. Rep., 2012. [Online]. Available: http://www.davidhbailey.com/dhbpapers/numerical-bugs.pdf

[17] IEEE, "IEEE Standard for Floating-Point Arithmetic (IEEE 754-2008)," IEEE, New York, Tech. Rep., aug 2008.

[18] B. W. Char, G. J. Fee, K. O. Geddes, G. H. Gonnet, and M. B. Monagan, "A tutorial introduction to maple," Journal of Symbolic Computation, vol. 2, no. 2, pp. 179–200, 1986.

[19] A. Griewank, "On Automatic Differentiation," in Mathematical Programming: Recent Developments and Applications. Kluwer Academic Publishers, 1989, vol. 6, pp. 83–108. [Online]. Available: http://www.researchgate.net/publication/2703247{_}On{_}Automatic{_}Differentiation/file/9c96052529013aed9e.pdf

[20] L. Hascoët and V. Pascual, "The Tapenade Automatic Differentiation tool: Principles, Model, and Specification," ACM Transactions On Mathematical Software, vol. 39, no. 3, 2013. [Online]. Available: http://dx.doi.org/10.1145/2450153.2450158

[21] C. Bischof, L. Roh, and A. Mauer-Oats, "Adic: an extensible automatic differentiation tool for ansi-c," Urbana, vol. 51, p. 61802, 1997.

[22] C. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland, "Adifor–generating derivative codes from fortran programs," Scientific Programming, vol. 1, no. 1, pp. 11–29, 1992.

[23] J. Utke, U. Naumann, M. Fagan, N. Tallent, M. Strout, P. Heimbach, C. Hill, and C. Wunsch, "Openad/f: A modular open-source tool for automatic differentiation of fortran codes," ACM Transactions on Mathematical Software (TOMS), vol. 34, no. 4, p. 18, 2008.

[24] M. Sagebaum, T. Albring, and N. R. Gauger, "High-performance derivative computations using codipack," arXiv preprint arXiv:1709.07229, 2017.

[25] R. J. Hogan, "Fast reverse-mode automatic differentiation using expression templates in c++," ACM Trans. Math. Softw., vol. 40, no. 4, pp. 26:1–26:16, Jul. 2014. [Online]. Available: http://doi.acm.org/10.1145/2560359

[26] F. S. Acton, Real Computing Made Real: Preventing Errors in Scientific and Engineering Calculations. Princeton, NJ, USA: Princeton University Press, 1996.

[27] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," ACM Trans. Math. Softw., vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2049662.2049663

[28] Y. Saad and M. H. Schultz, "Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems," SIAM J. Sci. Stat. Comput., vol. 7, no. 3, pp. 856–869, Jul. 1986. [Online]. Available: http://dx.doi.org/10.1137/0907058

[29] Y. Saad, "Ilut: A dual threshold incomplete lu factorization," Numerical Linear Algebra with Applications, vol. 1, no. 4, pp. 387–402. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/nla.1680010405

[30] A. Haidar, P. Wu, S. Tomov, and J. Dongarra, "Investigating half precision arithmetic to accelerate dense linear system solvers," in Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ser. ScalA '17. New York, NY, USA: ACM, 2017, pp. 10:1–10:8. [Online]. Available: http://doi.acm.org/10.1145/3148226.3148237

[31] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamari, "Rigorous Floating-Point Mixed-Precision Tuning," in Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'17). New York, NY, USA: ACM, 2017, pp. 300—-315.

[32] I. Karlin, J. Keasler, and R. Neely, "Lulesh 2.0 updates and changes," Tech. Rep. LLNL-TR-641973, August 2013.

[33] M. O. Lam and B. L. Rountree, "Floating-Point Shadow Value Analysis," in Proceedings of the 5th Workshop on Extreme-Scale Programming Tools, ser. ESPT '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 18–25. [Online]. Available: https://doi.org/10.1109/ESPT.2016.10

[34] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, and S. Tomov, "Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy," ACM Transactions on Mathematical Software (TOMS), vol. 34, no. 4, p. 17, 2008.

[35] X. S. Li, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Y. Kang, A. Kapur, M. C. Martin et al., "Design, implementation and testing of extended and mixed precision blas," ACM Transactions on Mathematical Software (TOMS), vol. 28, no. 2, pp. 152–205, 2002.

[36] H. Anzt, B. Rocker, and V. Heuveline, "Energy efficiency of mixed precision iterative refinement methods using hybrid hardware platforms," Computer Science-Research and Development, vol. 25, no. 3-4, pp. 141–148, 2010.

[37] M. O. Lam, J. K. Hollingsworth, and G. Stewart, "Dynamic Floating-Point Cancellation Detection," Parallel Computing, vol. 39, no. 3, pp. 146–155, mar 2013.

[38] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic optimization of floating-point programs with tunable precision," ACM SIGPLAN Notices, vol. 49, no. 6, pp. 53–64, 2014.

[39] F. De Dinechin, C. Q. Lauter, and G. Melquiond, "Assisted verification of elementary functions using gappa," in Proceedings of the 2006 ACM symposium on Applied computing. ACM, 2006, pp. 1318–1322.

[40] V. Magron, C. Verimag, G. Constantinides, and A. Donaldson, "Certified Roundoff Error Bounds Using Semidefinite Programming," ACM Trans. Math. Softw. Article, vol. 43, no. 34, 2017. [Online]. Available: http://dx.doi.org/10.1145/3015465

[41] E. Goubault and S. Putot, "Static analysis of finite precision computations," in International Workshop on Verification, Model Checking, and Abstract Interpretation. Springer, 2011, pp. 232–247.

[42] E. Darulova and V. Kuncak, "Towards a Compiler for Reals," ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 39, no. 2, pp. 8:1—-8:28, 2017.

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

[44] T. Braconnier and P. Langlois, "From rounding error estimation to automatic correction with automatic differentiation," in Automatic differentiation of algorithms. Springer, 2002, pp. 351–357.

[45] A. A. Gaffar, O. Mencer, W. Luk, P. Y. Cheung, and N. Shirazi, "Floating-point bitwidth analysis via automatic differentiation," in Field-Programmable Technology, 2002.(FPT). Proceedings. 2002 IEEE International Conference on. IEEE, 2002, pp. 158–165.

[46] A. A. Gaffar, O. Mencer, and W. Luk, "Unifying bit-width optimisation for fixed-point and floating-point designs," in Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on. IEEE, 2004, pp. 79–88.

[47] 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 Code Generation and Optimization (CGO), 2016 IEEE/ACM International Symposium on. IEEE, 2016, pp. 182–193.

[48] M. O. Lam and J. K. Hollingsworth, "Fine-grained floating-point precision analysis," The International Journal of High Performance Computing Applications, p. 1094342016652462, 2016.

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

[50] S. Graillat, F. Jézéquel, R. Picot, F. Févotte, and B. Lathuiliere, "Promise: floating-point precision tuning with stochastic arithmetic," in Proceedings of the 17th International Symposium on Scientific Computing, Computer Arithmetics and Verified Numerics (SCAN), 2016, pp. 98–99.

Appendix A
Proof of Analysis Accuracy

Given a set of inputs $\boldsymbol{S} \subset \mathbb{R}^n$ for some continuously differentiable function $\boldsymbol{f} : \mathbb{R}^n \rightarrow \mathbb{R}^m$, the following theorem provides a guarantee on the accuracy of the results produced by ADAPT for a given input $\boldsymbol{x} \in \boldsymbol{S}$.

Theorem: Let $\boldsymbol{f} : \mathbb{R}^n \rightarrow \mathbb{R}^m$ be a continuously differentiable function that takes as input the vector $\boldsymbol{x} \in \boldsymbol{S}$ and produces as output the vector $\boldsymbol{f}(\boldsymbol{x}) \in \mathbb{R}^m$. Let $\boldsymbol{y}$ be a lower precision representation of $\boldsymbol{x}$ such that $||\boldsymbol{x} - \boldsymbol{y}||_1 \leq \epsilon$ where $0 < \epsilon \leq ||\boldsymbol{x}||_1 \times 2^{-p}$ and where p is the number of mantissa bits and $||\boldsymbol{x}||_1 \times 2^{-p}$ is the absolute error bound (see Section III-A). Finally, let $\boldsymbol{J_f}$ denote the Jacobian matrix of $\boldsymbol{f}$, which gives the partial derivatives of the output with respect to the inputs. Then, there exists $L = \sup_{t\in[0,1]} \|\boldsymbol{J_f}\|$, such that $\|\boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{f}(\boldsymbol{y})\|_1 \leq L\epsilon$, where $L\epsilon$ is the specified error threshold (TolError in Algorithm 1).

Proof: By definition, $\boldsymbol{f}$ is locally Lipschitz and L is the corresponding Lipschitz constant. Define $\boldsymbol{J_f}$ as the $m \times n$ Jacobian matrix of $\boldsymbol{f}$, given by:

$$\begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

where $\frac{\partial f_i}{\partial x_j}$ is the partial derivative of the i^{th} component of the output with respect to the j^{th} component of the input.

For some $\boldsymbol{y} \in \mathbb{R}^n$, define $\boldsymbol{z}(t)$ such that for $\boldsymbol{x} \in \boldsymbol{S}, \boldsymbol{z}(t)$ satisfies $\boldsymbol{z}(t) = \boldsymbol{y} + t(\boldsymbol{x} - \boldsymbol{y})$, $0 \leq t \leq 1$. Clearly, $\boldsymbol{z}(0) = \boldsymbol{y}$ and $\boldsymbol{z}(1) = \boldsymbol{x}$. From the fundamental theorem of calculus and chain rule of differentiation, we have that

$$\begin{aligned} \boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{f}(\boldsymbol{y}) &= \boldsymbol{f}(\boldsymbol{z}(1)) - \boldsymbol{f}(\boldsymbol{z}(0)) \\ &= \int_0^1 \boldsymbol{f}'(\boldsymbol{y} + t(\boldsymbol{x} - \boldsymbol{y}))(\boldsymbol{x} - \boldsymbol{y})dt \\ &= \int_0^1 \boldsymbol{J_f}(\boldsymbol{x} - \boldsymbol{y})dt \end{aligned}$$

It follows that

$$\|\boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{f}(\boldsymbol{y})\| = \|\int_0^1 \boldsymbol{J_f}(\boldsymbol{x} - \boldsymbol{y})dt\|$$

Using the triangle inequality for integrals, we get

$$\begin{aligned} \|\boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{f}(\boldsymbol{y})\| &\leq \int_0^1 \|\boldsymbol{J_f}(\boldsymbol{x} - \boldsymbol{y})\|dt \\ &\leq \sup_{t\in[0,1]} \|\boldsymbol{J_f}\|\|(\boldsymbol{x} - \boldsymbol{y})\| \int_0^1 dt \\ &\leq L\|\boldsymbol{x} - \boldsymbol{y}\| \\ &\leq L\epsilon \end{aligned}$$