Tuning Floating-Point Precision Using Dynamic Program Information and Temporal Locality

Hugo Brunie, Costin Iancu, Khaled Z Ibrahim, Philip Brisk, Brandon Cook

Tuning Floating-Point Precision Using Dynamic Program Information and Temporal Locality

Hugo Brunie*, Costin Iancu*, Khaled Z. Ibrahim*, Philip Brisk†, Brandon Cook*
*Lawrence Berkeley National Laboratory
{hbrunie,cciancu,kzibrahim,BGCook}@lbl.gov
†University of California, Riverside
philip@cs.ucr.edu

***Abstract*—We present a methodology for precision tuning of full applications. These techniques must select a search space composed of either variables or instructions and provide a scalable search strategy. In full application settings one cannot assume compiler support for practical reasons. Thus, an additional important challenge is enabling code refactoring. We argue for an instruction-based search space and we show: 1) how to exploit dynamic program information based on call stacks; and 2) how to exploit the iterative nature of scientific codes, combined with temporal locality. We applied the methodology to tune the implementation of scientific codes written in a combination of Python, CUDA, C++ and Fortran, tuning calls to math exp library functions. The iterative search refinement always reduces the search complexity and the number of steps to solution. Dynamic program information increases search efficacy. Using this approach, we obtain application runtime performance improvements up to 27%.**

I. Introduction

Reducing the floating point precision of data in scientific applications has been shown [2], [12], [15] to lead to performance improvements due to reduction in data movement or computational intensity. Colloquially, the goal is to replace the maximum number of operations with high precision operands (e.g. `double`) with a lower precision (e.g. `single`) version.

Previous work in the area can be categorized as algorithm specific [12] and algorithm independent (generic). The former is based on the observation that many algorithms are naturally iterative. The algorithm-specific implementation starts in low precision, and then patches and transitions to a higher precision as the algorithm proceeds. The latter use program analysis tools to tune arbitrary pieces of code. They select a program search space and then search it using a principled methodology that adjusts either program variables [28] or assembly instructions [18]; the main challenge is to reduce complexity.

Our objective is to transform arbitrary codes where individual functions can be replaced with multiple implementations using different precision variables and operations, resulting in different performance and correctness profiles. For example, any code that uses generic hardware-accelerated math functions could benefit from a principled approach to select the appropriate implementation each time the function is called. We anticipate that this problem will become increasingly prevalent over the next decade, given current trends for hardware specialization in High Performance Computing (HPC).

We argue in favor of a generic performance-driven methodology to tune the precision of full-fledged applications. To maximize the appeal of our approach to HPC practitioners, we assume no compiler support, while providing the ability to refactor code to accommodate mixed precision. We also designed and implemented heuristics to reduce the complexity and overhead of our search procedure, which explores the transformation space using a combination of dynamic program behavior and temporal locality information. This contrasts with existing techniques that rely primarily on static program analyses to guide precision tuning, and thereby lack a clear path to incorporate performance feedback into the tuning strategy.

Our system executes an application to generate a trace of the operations of interest, annotated with time and precision-related floating-point accumulation error. It tracks and tunes precision on the granularity of function invocations, reasoning on "instructions". The system first *identifies structure within the trace that is associated with code properties and enables refactoring* **and** *devises a scalable search strategy that uses refinement based on "structure" determined across multiple dimensions.* The paper describes several refinement methods and evaluates each in terms of its impact on search complexity (i.e., the number of trials) and efficacy (i.e., either a measured performance improvement or an increase in the number of calls to reduced-precision implementations of functions chosen for tuning.).

Search Space Classification: The simplest classification method uses static source code information, such as the line number of a call to a function. This space can be further refined using two criteria: 1) control flow information; and 2) temporal locality. Refinement using control flow information recognizes that calls to functions during program execution are not arbitrary, but depend on the structure of the program itself, encompassing each function's control flow graph and the static locations of function call sites. Recognizing feasible and likely call

sequences enables community formation; we define a community to be a sequence of commonly-executed dynamic function calls whose precision can be tuned en masse. The process of community[10], [9], [33] formation, which we refer to as clustering, points toward a hierarchical [11] methodology to explore the search space.

Refinement using temporal locality provides an orthogonal criterion for further classification and search. Program behavior evolves over time, thus clustering based on static criteria or "averaging" across the whole execution may lead to the formation of large communities. Tuning at large granularity may lead to diminished success. Thus, we argue for refining the tuning process using temporal information about the program execution.

To enable temporal refinement, our system associates each static source location with its call stack (backtrace) at each dynamic invocation. This increases the complexity of the search (more primitives to examine), but adjusts the granularity of the search so that it can consider program behavior over time. This offers the potential to improve efficacy while leveraging backtrace information to guide the refactoring process.

Iterative Search Refinement: In our system, a dynamic function call is an event of interest. Our approach to community formation yields a hierarchical representation, in which events are clustered to form larger communities. As mentioned earlier, precision tuning can be performed on the granularity of communities, motivated by the observation that a community will often correspond to a sequence of iterations of a loop or loop nest in the source code of an HPC application; we expect this to simplify code refactoring. Current precision tuning systems [28], [11], [2], [16], [15] employ a binary search akin to delta debugging [34]; the drawback of these approaches is that there is no way to predict, a-priori, the impact of a decision to reduce precision on overall application performance. Our hierarchical representation allows rapid identification of the communities that maximally impact performance, which allows our tool to apply a faster and more efficient greedy linear search to make precision tuning decisions.

Additionally, we advocate for an iterative refinement approach to prune the search space: our tool starts by tuning clusters of events classified by their source code location; it then tunes individual events, reclassifies them, and forms communities based on backtrace information; lastly, it tunes precision over sets of backtraces. Intuitively, this approach is designed to make as much progress as possible using the coarsest representation available before moving to use more precise information.

Validation and Impact: We validate our approach on representative HPC applications that make heavy use of transcendental functions such as `exp`: the nanoBragg spots simulation [22], which is part of a larger Computational Crystallography Toolbox (CCTBX) [5], and PeleC [27], [6], an adaptive-mesh compressible hydrodynamics code for reacting flows. These codes employ a combination of Python, C++, Fortran, and CUDA. Between the two, CCTBX has a simpler software engineering structure, and static classification with clustering suffices to discover most of the tuning potential. We lower 100% of the total calls for the CUDA kernel, resulting in performance improvement of 27%. For PeleC (PMF 2D), which has a more complicated code structure, static selection uncovers seven call sites, and a total of 151 calling contexts in C++ and Fortran. Here static classification lowers 70.7% of the calls, while a hybrid static+backtrace search improves it to 79.9%.

The work indicates the need for incorporating dynamic program information (call stack and temporal locality) in the tuning of floating point precision for full applications. The methodology is directly applicable to other codes that call multi-versioned expensive scalar operations (e.g. `sin`), and as discussed in Section IX is extensible and composable with other formulations.

The rest of this paper is structured as follows. In Section II we discuss the motivation behind this effort. In Sections III,IV and V we describe the design of the search approach: search space classification, clustering and search algorithm respectively. In Section VI we describe the implementation and Section VII our application benchmarks. Detailed results are discussed in Sections VIII and IX.

II. Motivation

Our primary motivation to study precision tuning is to enable HPC software developers to refactor their code to maximally leverage the performance benefits of hardware specialization. The basic premise is that it is relatively easy for HPC library developers to provide multiple implementations of arithmetic functions to support varying precision, but it is much harder to know which version of the function to use when developing an application. In existing systems, this approach is often implemented by the use of intrinsic operations associated with common mathematical functions such as `exp`, `sin` etc. For example, a call to `exp()` in single or double precision shows as much as ×1.5 performance difference on an Intel Haswell CPU, and ×2 speedup on NVIDIA Volta GPU.

With the demise of Moore's Law, it is anticipated that future hardware performance improvements will be obtained through specialization, such as fixed function units that manipulate narrow data types. One such example is the NVIDIA Volta V100, which offers FP32/FP16 precision 4x4 matrix multiply units. NVIDIA GPUs featuring Tensor Cores will be deployed on all large scale US Department of Energy (DOE) installations under procurement in the 2020-2022 time frame. This announcement has already motivated work in algorithm-specific refactoring [12] for mixed precision DGEMM, which reduces full computation in double-precision to a half-precision product coupled with single-precision accumulation. Along similar lines, we expect there to be growing interest in refactoring applications to utilize reduced-precision data types and

operations for hardware acceleration, as many vendor roadmaps contain specialized functional units.

```
10
# Function call to nanoBragg in Python
12 SIM = nanoBragg( ... )

14 # flux is always in photons/s
SIM.flux=1e12
16 ...
#outer loop in Python
18 for x in range(len(flux)):
    SIM.add_nanoBragg_spots_cuda()
20
    // C++ loop nest in a 300-line CUDA kernel
22  // that computes CCTBX nanoBraggSpots
    for(spixel=0;spixel<spixels;++spixel) {
24      for(fpixel=0;fpixel<fpixels;++fpixel) {
            ...
26          if(xtal_shape == GAUSS)
                // fudge the radius so that
28              // volume and FWHM are similar
                // to square_xtal spots
30              F_latt = Na*Nb*Nc*
                    exp(-(hrad_sqr/0.63*fudge));
32      }
     }
```

Fig. 1: *The nanoBragg computation kernel, depicting one call site of the exp() function. We successfully tuned the precision of this function using the methodology outlined in this paper.*

Figure 1 depicts the nanoBragg spots computation kernel, which is part of CCTBX. An outer-loop, written in Python interacts with a 300-LoC CUDA kernel, which includes a loop nest written in C++, which includes a call to `exp(double)`. By tuning the precision of this call we mean replacing it with a call to the `__expf(float)`, which provides lower precision but better performance. The decision has to be valid across all inputs and execution paths within the application, meaning that it cannot produce outputs unacceptable to users. This particular snippet also illustrates the practical challenges when considering full applications, which are often composed of multiple programming languages.

III. Search Space Selection

Existing tools that tune precision either search over program variables [28], [29], [11] or instructions [18], [17]. These approaches are generally limited to tuning scalar variables precision and have not (yet) demonstrated scalability beyond small programs. Additionally, they require compiler tool-chain support. Our approach, in contrast, targets manual code transformation and aims to scale to large code bases that use many variables, including large arrays.

To meet these objectives, our software searches the instruction/invocation space generated during application execution. It extracts program structure from execution traces and relates this structure to constructs in the program source code to enable manual refactoring. Once again, the focus here is multiversioned functions, in which several implementations are provided using data types of varying precision. Based on the output of our tool, the application developer can make an informed decision to rewrite the program to call different implementations of the function, and under what condition(s) each call will be made.

An informal statement of the precision tuning problem is the following: given the sequence of calls to a multi-versioned function, select a maximal subset of calls to execute at reduced precision, while maximally improving performance and ensuring that the accumulated error does not exceed a given threshold.

A. Search Space Characterization Using Control Flow

Figure 2 depicts a graphical representation of the sequence of calls to the `exp` function during the execution of PeleC, one of our chosen benchmark applications. The x-axis shows the timestamp of the call, while the y-axis corresponds to the static source location (SLOC) of the call, whose line number in the source code is known. Figure 2 depicts both dimensions of our precision tuning approach: community formation along the y-axis represents the usage of static control flow information at call sites, while community formation along the x-axis represents the opportunity to leverage temporal locality information, e.g., repeated calls to a function during the iteration of a loop.

There are seven static call sites to `exp` in the PeleC source code. In figures 3 and 4 these call sites are color coded Blue (B), Green (G), Red (R), Purple (P), Brown (Br), Gray (Gr) and Pink (Pi). Previously proposed approaches [28], [16] try to *lower the precision of all calls (assembly instruction) at a given static source location.* This amounts to projecting all the points on the y-axis in the figure and searching over seven call sites, i.e. 128 combinations *Search(B, G, R, P, Br, Gr, Pi).*

More recent approaches [2] exploit the iterative nature of scientific codes and pose the problem as a binary search over the execution trace of the function invocations, without any further distinction between invocations. Effectively, this projects all of the points on the x- axis and searches over all ($\approx 10^7$) function calls, $Search(1, \ldots, 10^7)$. This approach lacks the ability to correlate precision tuning decisions with source code locations, complicating, or, in the worst case, rendering infeasible, the programmer's ability to refactor the code.

Our proposed strategy is to exploit control flow information to identify structure, which we then use to reduce the tuning search space. Figure 3, for example, shows the first 80,000 events and clearly visualizes similarities and differences between the frequency and sequences of calls from different sites. In this case, our strategy is to form three communities and search them independently: $Search(G, B)$, $Search(R, P)$ and $Search(Br, Gr, Pi)$.

Our search procedure, which we will describe in Section V, has a combinatorial (exponential) time complexity, but can run as fast as super-linear in the best case. Given these parameters, we believe that our search procedure is best implemented in conjunction with a scalable and hierarchical community formation strategy. Section IV-A

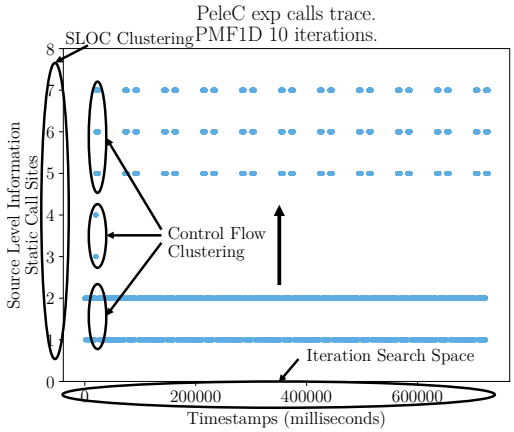

Fig. 2: *Plot of all* `exp` *calls in PeleC.*

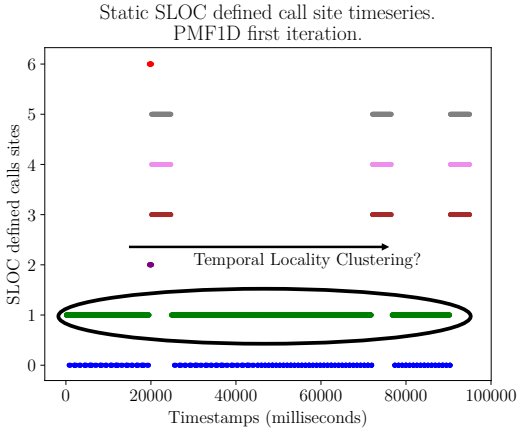

Fig. 3: *Finer grained structure of* `exp` *calls in PeleC. This structure enables clustering. Execution trace suggests temporal locality as an orthogonal clustering criteria.*

introduces an algorithm to uncover clusters of the call sites of a function of interest using control flow information.

B. Search Space Characterization Using Temporal Locality Information

Thus far, we have shown how to incorporate temporal information to perform community formation on the granularity of individual call sites. To further prune the search space, we desire an approach to community format that can dynamically vary the precision of repeated calls from the same site. Figure 3 suggests that there exists a temporal locality component to the data, which we postulate is due to the iterative nature of PeleC, as well as many other scientific codes.

Our approach is to integrate methods from time series analysis to the clustering process. *We propose to observe the evolution of the program (stack) backtrace in order to better understand its temporal behavior.* Figure 4 depicts 62 backtraces associated with the original seven SLOCs. Each backtrace is color coded with a gradients to depict the mapping to the static source code location.

To further reduce dimensionality, community formation can be applied to backtrace data in the control flow dimension. Regardless, this does yield a larger search space to explore compared to classification by SLOC, but the fine-grained temporal decomposition offers the possibility to obtain a better overall solution.

> Design Alternative: How should the tuning procedure use static (SLOC) and dynamic (backtrace) classification?

IV. Trace Data Classification and Clustering

Community detection is often formulated using graph or network theory [10], [9], [33]. The first step is to derive a graph from a backtrace obtained from dynamic execution of a program. Our objective is to capture both control flow and temporal behavior.

Let $S = \{s_1, s_2, \ldots, s_m\}$ be the set of call sites[1] to one or more functions whose precision we aim to tune, and let $T = \langle t_1, t_2, \ldots, t_n \rangle$, $t_i \in S$ be the trace, i.e., the dynamic sequence of calls. In a typical HPC application deployed at scale, one can typically assume that $m \ll n$. We define a function $f : T \to S$ to establish a mapping between dynamic calls and call sites; in other words $f(t_i) = s_j$ if the i^{th} dynamic call in the trace occurs at the j^{th} call site.

We define $U_{ij} = \{(t_k, t_{k+1}) \in T^2 | f(t_k) = s_i \wedge f(t_{k+1}) = s_j\}$ to be the subset of consecutive calls at sites s_i and s_j in T. The set U_{ij} captures control flow through the identification of consecutively occurring events in T. To capture temporal locality along with control flow, it is possible to further filter the data using a tunable parameter Δ. We define

$$U_{ij}^{\Delta} = \bigcup_{l=1}^{\Delta} \{(t_k, t_{k+l}) \in T^2 | f(t_k) = s_i \wedge f(t_{k+l}) = s_j\}. \quad (1)$$

to be the subset of calls to site s_j that occur after up to Δ calls following an occurrence of a call at site s_i.

A backtrace graph is a weighted directed graph $G^{\Delta}(S, E^{\Delta}, w)$, where $w : E^{\Delta} \times E^{\Delta} \to N$, $E^{\Delta} = \{(s_i, s_j) || U_{ij}^{\Delta}| > 0\}$, and for edge $e = (s_i, s_j)$, $w(e) = |U_{ij}^{\Delta}|$. In other words, edge e exists in E^{Δ} if at least one call at site s_i is followed by a call at site s_j within the next Δ calls in the dynamic trace, and $w(e)$ is the number of times that this occurs.

Community detection can then be performed on the backtrace graph. One option is to use the Girvan-Newman algorithm [10], which runs in $O((E^{\Delta})^2 \times S)$ time complexity. Other community detection algorithms that exhibit tradeoffs between accuracy and computational efficiency

[1]For simplicity, we assume individual call sites here. In practice, the s_i's will correspond to a mixture of call sites and backtraces, depending on application behavior.

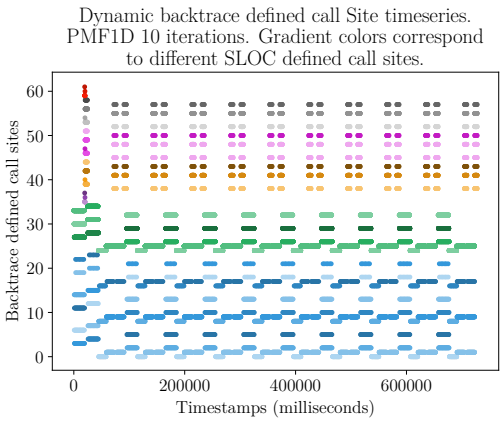

Fig. 4: *Dynamic backtrace classification refines classification in the temporal locality direction*

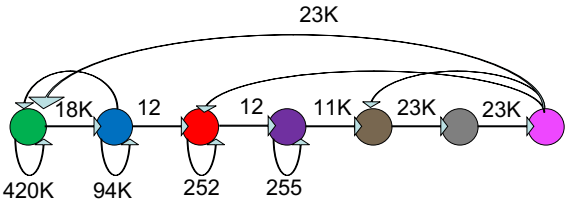

Fig. 5: *Graph associated with the example PeleC trace. Each node represents the call source location. Weights on edges represent the number of transitions. For community detection algorithms we build a graph with as many edges as the weight shown.*

also exist [9], [33]. Guo et al [11], for example, use modularity maximization to perform a data flow-centric variable clustering; however, according to one recent survey [21] modularity maximization offers performance advantage, while sacrificing accuracy. Regardless, our approach is not tied to any specific algorithm for community detection.

The clustering obtained with Girvan-Newman community detection algorithm depends on Δ. This illustrates both the power and the challenges of using generic community detection algorithms.

A. Simplified Clustering Algorithm

The control flow graphs in our application space are neither large nor exceptionally complicated in terms of structure. Consequently, we can perform a simple, linear-time community detection heuristic that eschews the intricacies of more complicated schemes such as Girvan-Newman. Our approach runs a cycle-detection algorithm on the event graph and then sorts cycles by length. Simple cycles are clustered (i.e., identified as communities) and are then deleted. This process transforms at least one composite cycle, if one exists, into a simple cycle; the process repeats until no cycles remain. Any acyclic connected components in the remaining graph become communities in and of themselves. In the PeleC example, this procedure yields communities $(B, G), (R, P), (Br, Gr, Pi)$.

B. Using Static Information to Guide Backtrace Clustering

When considering backtrace data, one choice is to perform clustering on the raw trace data.

However, a backtrace is uniquely associated with the source code location of the event. *Thus, we can always identify a community based exclusively on SLOCs for any cluster of backtraces that our algorithm identifies.* We exploit this observation to provide a prefiltering stage for the backtrace clustering. We first identify communities based exclusively on SLOCs of call sites $S = \{s_1, s_2, \ldots, s_m\}$; in our example, this will yield communities $(B, G), (R, P), (Br, Gr, Pi)$. We then build the graphs for backtrace data associated with each cluster. In this example we will build three backtrace graphs, each independently passed into another clustering stage.

> Design Alternative: What is the best clustering strategy? Should backtrace data be preclustered using static information?

V. Search Algorithm

The search algorithm tunes the precision of each event to determine which operations, variables, etc. can be executed/represented in low precision. Informally, the input is a set of events and their possible choices. e.g. $\{(e_1^f, e_1^d), \{(e_2^f, e_2^d)\}..\{(e_N^f, e_N^d)\}$, where f and d represent a choice between single and double precision. The solution of the search is an instantiation of the event whose precision is no greater than the original program (hopefully lower) which satisfies one or more acceptability criteria in terms of floating-point error. The output of the search has the form $\{e_1^{q_1}, e_2^{q_2}, \ldots e_N^{q_N}\}$, where $q_i \in \{d, f\}$, $1 \leq i \leq N$.

At each step, the program is executed using the candidate solution, and is checked against the user-specified acceptance criteria. In our case, we classify events based either on their SLOC s_i, their SLOC-backtrace pair $p_i = (s_i, bt_{i,j})$, or the community C_i to which s_i or p_i belongs. In the subsequent discussion, we use the general term "site" to refer to either of these three options, and we use the C_i notation, as it is the most general.

The optimal solution requires a brute force search of all possible combinations, while most prior research effort in precision tuning explores scalable heuristics [28], [18], [2], [15]. The following subsections respectively introduce a scalable search procedure along with techniques to prune the input space.

a) Search Procedure: Our search algorithm is presented in Algorithm 1, procedure *Search*(..). The input is a list of sites; the search proceeds in two stages: initial

filtering of unlikely candidates, followed by a brute force search over the remaining candidates. At each stage the search prioritizes the evaluation of items in the order of their performance impact. The initial filtering tries to lower the precision of each candidate in isolation. If the program fails to meet correctness criteria, the candidate is discarded for the next phase, which performs a *breadth first search* over all possible tuning options for input events; the search tends to converge quickly because the combinations to consider are ordered based on performance impact a-priori.

Breadth first search has scalability concerns in the general case; however, our choice is motivated by two mitigating factors: 1) the desire to increase the robustness of the solution, since application developers are likely to be reluctant to otherwise accept our proposed transformations; 2) tight control over the number of inputs using the incremental input filtering procedure, described below. We can trivially replace our algorithm with any other search strategy from the literature.

Algorithm 1: Reduced Precision Heuristic.

```
Function Search(X)
    Input: X: set of sites
    Output: S: successfully lowered sites, F: failed to lower sites
    F, S ← ∅ ;
    /* Forward phase: compute individual sites                    */
    foreach x ∈ X do
        if ReducePrecision(x) then
            S.extend(x) ;
        else
            F.extend(x) ;
    /* Backward phase: search multi-site                          */
    Y ← Combinations(S) ;
    K ← ExecutionCost(Y) ;
    L ← sort(Y, K);
    S ← ∅ ;
    while not L.isEmpty() do
        x ← L.pop(0) ;
        if ReducePrecision(x) then
            S.extend(x) ;
            F.extend(¬x) ;
            return S, F;
    return S, F;
Function IncrementalFilteringSearch(τ_{l,s}, Δ)
    Input: τ_{l,b}: Time series for SLOC site l and backtrace site b
           Δ: clustering time threshold
    Output: S: Reduced precision sites
    S ← ∅ ;
    /* Projection into SLOC dimension                             */
    L ← {l : |τ_{l,:}| > 0};
    /* Build a graph for SLOC sites                               */
    G ← BuildGraph(L, τ_{l,b}, Δ ) ;
    /* Cluster sites to reduce search space                       */
    C ← ClusteringAlgorithm(G) ;
    /* Find sites for precision lowering                          */
    S_c, F_c ← Search(C) ;
    S.extend(S_c) ;
    /* Expand failed clusters to SLOC sites                       */
    L_F ← {l : l ∈ F_c};
    /* Search while lowering prior discovered sites               */
    S_l, F_l ← Search(L_F) ;
    S.extend(S_l) ;
    ...;
    /* Expand failed SLOC sites based on backtrace                */
    B ← {b : |τ_{:,b}| > 0 & b ∈ F_l};
    G ← BuildGraph(B, τ_{l,b}, Δ ) ;
    C ← ClusteringAlgorithm(G) ;
    S_b, F_b ← Search(C) ;
    S.extend(S_b) ;
    return S ;
```

b) Incremental Filtering: The procedure *Search*(..) performs the first stage of filtering, which discards any sites that cannot be individually lowered during the brute force search over combinations of sites. More aggressively filtering can be implemented based on the arity of the initial search space produced by the different classification criteria discussed in Section III. This is based on the observation that a number of vertices in a graph is an upper bound on the number of communities that can be identified, and there will be fewer SLOCs than (SLOC, backtrace) pairs.

Procedure *IncrementalFilteringSearch*(..), described in Algorithm 1, applies this principle. The first stage searches over the relatively small number of clusters formed exclusively from SLOC; any solution discovered is applied to the program, and any SLOCs contained in the cluster (SLOC-C) are removed from further consideration during the search. The second stage searches over the remaining SLOC sites, which are far fewer than the number of (SLOC, backtrace) pairs; once the program is updated, the SLOCs identified here are removed from further consideration. The next two stages of the search are applied to clusters formed exclusively from backtraces (BT-C) and the remaining backtraces (BT).

The choice to search clustered sites prior to individual sites is intuitive; the choice to first search the SLOCs, which are statically known, prior to (SLOC, backtrace) pairs, which are obtained using dynamic information, is arguably an application-dependent choice.

> Design Alternative: What is the best composition of iterative filtering?

VI. Implementation and Validation Strategy

Our implementation of the search procedure uses link time interposition for portability and generality. The set of functions of interest are wrapped with code that enables either profiling or their lowering based on the classification criteria: SLOC or backtrace. We note that handling multiple languages and compilers can be challenging and our implementation was validated using only the `GNU` compiler toolchain. The tool can be downloaded on github: https://github.com/hbrunie/PyFloT.

In the first stage we profile the application to determine the potential for performance improvement. We then generate the execution traces. For the astute reader, we note that tracing affects timings and if pure temporal based clustering is desired we make the assumption that the tracing overhead per event is a constant. The trace data contains both SLOC and backtrace information. Our framework is composed of a C++ library, using GOTCHA [26] framework to capture backtraces.

The tracing infrastructure has not been optimized for speed and we currently observe ×100 slowdown. One easy

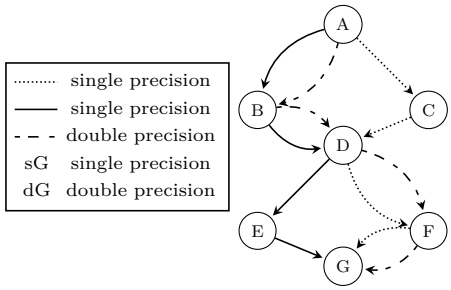

Fig. 6: *Non-modified control flow.*

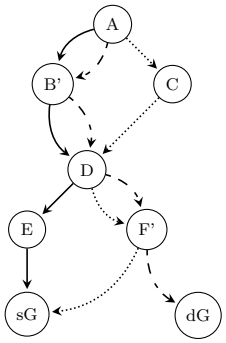

Fig. 7: *Minimizing number of modifications.*

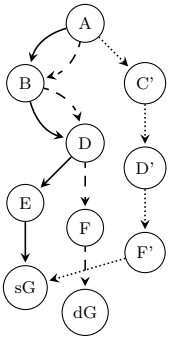

Fig. 8: *Minimizing branching performance impact.*

way to lower tracing overhead is to add sampling. Sampling can be added manually at the macro application level by deciding which high level iterations of an algorithm/method to observe. We have successfully applied this technique in our experiments for the initial exploration. Sampling can be also applied at the micro-level, by recording individual events according to a "frequency" (with decay).

We then analyze the traces to generate the input sets for the search procedure. This involves counting all the events associated with a SLOC or backtrace, building the graphs necessary for clustering and building the clusters. Each step of the search involves running the program with the desired set of calls in lower precision. Again, this is performed using link interposition and it is fully automatic, open source, with a python code implementing algorithm 1.

A solution is found based on user acceptability criteria. We note that this turned out to be a challenging step, where we reached out to the application developers and actually ignited a debate within their community. Based on their feedback, this interaction benefited application development as it forced them to formalize the mathematical acceptability criteria.

We run the procedure on multiple inputs obtained from the application developers and report the final solution as the intersection of all solutions for each input.

By correlating the number of events lowered in the traces with the initial unmodified application profiling we get an estimate of the potential performance improvements.

The last step of the process is the application refactoring, where we modify the source of the application to perform the desired calls in lower precision.

A. Application Source Code Refactoring

Modifications for solutions that include SLOC sites are trivial, due to our decision to lower all events associated with a site. The decisions to lower sites associated with particular backtraces are more complex and application specific. Without compiler support we cannot develop an automated procedure (i.e. precisely identify interprocedural loops and/or branches), but we provide the basic guidance to determine the minimal number of code changes. There are two guiding principles: 1) we want to minimize the number of changes; and 2) we want to minimize the number of branches introduced and in particular avoid fine grained per call branches.

Consider the call graph presented in Figure 6, where the paths ACDFG (densely dotted) and ABDEG (solid line) lead to the leaf G and can be executed in single precision. Whereas, when the program takes the path ABDFG (dash pattern), G needs to be executed in double precision. In this example the path describes the backtrace. The minimal set of nodes to modify must contain at least enough nodes to differentiate the different precision paths. Here F and B make the difference between, respectively, ABDFG from ABDEG and ABDFG from ACDFG. Thus the minimal set is F, B.

The best starting point for refactoring is examining the set of functions F, B. There are multiple strategies available, depending on the identification of loop and branch structures in the code and user software engineering and maintenance concerns. One possible strategy is marking the path and adding control divergence checks in dominator nodes for paths that share function calls. This is illustrated in Figure 7, where we mark the execution of B' and decide to specialize F'. Another strategy is cloning nodes across the path, as illustrated in Figure 8.

VII. Benchmark Results

A. Experimental Setup

All experiments ran on the Cori supercomputer at the National Energy Research Scientific Computing Center (NERSC). PeleC results were obtained on the 1.4 GHz Intel Xeon Phi Processor 7250, code vectorized with AVX-512, MCDRAM configured in cache mode. CCTBX results were obtained on a 2.4GHz CPU Intel(R) Xeon(R) Gold 6148 CPU coupled with GPU Tesla V100-SXM2, 16GB, using CUDA 9.2. The speedup is measured with nvprof.

B. PeleC

Pele is a suite of adaptive mesh hydrodynamics simulation codes for reacting flows [27]. It provides a toolkit of methods which are used by other domain science codes. Pele contains over 3 Million lines of Python, C++ and Fortran code.

For our study we consider the Pele Combustion Suite (PeleC), which allows the user to select different equation of state (eos) as the constitutive equation and close the compressible Navier-Stokes system of equations. PeleC solves the reacting compressible Navier-Stokes on a structured grid, optionally with embedded boundary geometry treatment and non-ideal gas equations of state. It performs time-dependent, adaptive mesh refinement (AMR) for large ranges of spatial and temporal scales in turbulent reacting flow.

a) Benchmarks: PeleC allows multidimensional formulations of the simulation of interest. For this study we have selected the Premixed Flame tests, in their 1D and 2D formulations. These are referred to as PMF 1D and PMF 2D.

b) Acceptability and Inputs: The acceptability criteria for PMF is the speed of the premixed flame, with the expectation it reaches an asymptotic value when simulating for a large number of time steps. Figure 9 shows the flame speed for PMF 1D against timesteps from 0 to 60,000. For our purposes, we were interested in determining a relative error threshold for this final value, which spurred a long debate within the PeleC developer community. Their initially recommended bound for the absolute error was set to $1E-6$, to account for the of the number of significant digits (4-6) in the input data extracted from real life experiments. Our experiments ran with this value, but we note that since they relaxed their recommendation to 10% relative error.

Our results have been validated over 10 inputs each for PMF 1D and PMF 2D. For each input we adjust the "pressure" parameter with random values between 1Pa and 10Pa.

c) Application Profile: PMF 1D spends 16% of its execution time in `exp`, while PMF 2D spends only 5%, but it is the second greatest hotspot, the first one taking 8%. These amount to 164K and 75M calls respectively. In both applications `exp` is called from the same seven static source locations. Two of these are in Fortran code `egz_module.f90`, the rest of five are in the `LiDryer.cpp` generated C++ code. In PMF 1D, there are 62 different backtraces associated with the seven static source locations, while in PMF 2D we observe 151 backtraces.

For reference, there are total 6 `exp` call sites inside `egz_module.f90` and 254 in `LiDryer.cpp`, a chemistry model generated automatically by PelePhysics Python code.

d) Summary of Performance Results: For brevity, we summarize the performance results for PMF 1D, while we provide a more detailed description for PMF 2D in Section VIII. In PMF 1D we are able to lower all the calls associated with six source locations, which amounts to lowering 99% of the total number of calls during execution. This translates into 8% execution time improvement. For PMF 2D we are able to lower calls associated with the same static locations, which amounts to lowering 78% of the calls during execution. This amounts to roughly 3% performance improvement. Figure 10 shows that our transformations still produce output within the accuracy constraints communicated by the PeleC developers.

e) Source Code Refactoring: For any static source location identified by the analysis we replace `exp` with `expf` in the C++ code and `exp(x)` with `exp(real(x,kind=4))` in the Fortran code.

PeleC provides a very good illustration of the practical challenges encountered when attempting full application precision tuning. For the Fortran call sites developers are willing to consider the changes. For reference, the Fortran calls amount to 75% of the total calls in PMF 1D and for 30% in PMF 2D. They are reluctant to tune and accept changes in the automatically generated C++ code. Thus for acceptability in those cases, we shall need to consider the refactoring of the PelePhysics code generators. Furthermore, the code associated with the dynamic backtrace is C++, complicating adoption and refactorization.

C. CCTBX: nanoBragg spots

The Computational Crystallography Toolbox (CCTBX) [5] is being developed as the open source component of the PHENIX [20] system. Its `simtbx` module is a Simulation Toolbox for simulating X-ray diffraction images ab initio. The `cctbx` module contains libraries for general crystallographic applications, useful for both small-molecule and macro-molecular crystallography. CCTBX contains roughly 600K lines of Python, C++ and CUDA code.

a) Benchmarks: For this study we have selected the nanoBraggspots computation kernel [22] from the `cctbx` module, called from `simtbx`. The code from the CCTBX test suite performs the simulation of a crystal specified by input parameters.

b) Acceptability and Inputs: The output of the simulation is a $3,000 \times 3,000$ pixel image, as shown in Figure 11. While they could not give an exact criteria derived from experimental data or first principles as in the PeleC case, the CCTBX developers stated they will accept differences in few image pixel. In our tuning we aim for at most one pixel difference from the 9,000,000 image pixels, which has been accepted by the CCTBX developers. The magnitude of the difference was not mandated, but this may change as CCTBX is evolving at a rapid rate. The input data is a set of parameters characterizing the crystal structure and the CCTBX regression test suite includes generators for uniformly random parameters. Application developers stated that a correct result on ≈ 10 inputs would convince of the

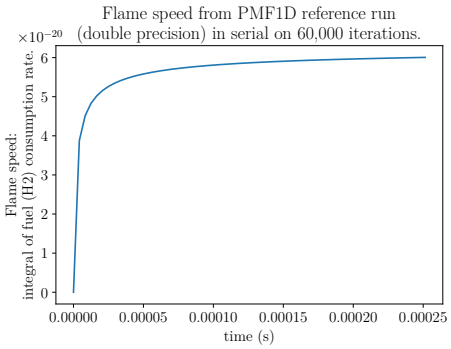

Fig. 9: *PMF 1D Premixed Flame speed against time steps.*

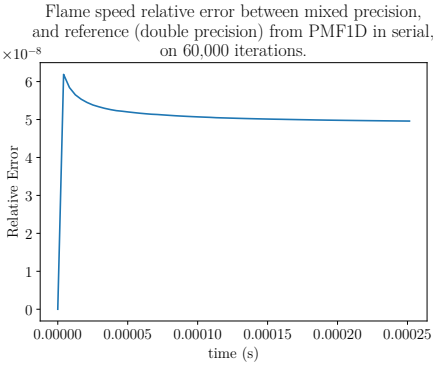

Fig. 10: *PMF 1D Premixed Flame speed: Relative error comparing mixed precision to reference run (double precision).*

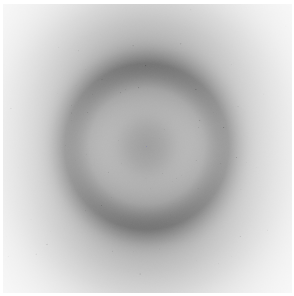

Fig. 11: *9M pixels image obtained after execution of the nanoBragg spots simulation (CCTBX).*

stability of the computation. Our results are validated on 20 different generated inputs.

c) Application Profile: nanoBragg spot simulation spends 14% of the execution time in `exp` calls and we observe four static call sites in the CUDA code.

d) Summary of Application Results: nanoBragg exhibits a simple dynamic behavior with 100% executions of `exp` coming from a single call site. The analysis is able to lower this site, resulting in kernel execution time improvement of 27%. Looking into the superlinear speedup we made two observations: (i) The NVPROF PC samples do not count the latency of each instruction, thus it is not accurate (ii) replacing `exp` with `__expf`, more values are truncated to 0. We suspect truncation simplify some computation inside GPU floating point arithmetic units as we observe a high ratio of reduced PC sampled on unchanged lines of code: e.g. 2000% reduction in a line executing a condition branching, and 1200% in the line computing the final pixel intensity.

e) Source Code Refactoring: We replace the original calls to `exp` with the CUDA `__expf` call. The change is in discussion to be adopted in the official CCTBX source code.

VIII. Analysis of the Search Evolution

We judge the efficacy of the search algorithm along several criteria: 1) number of trials required to attain an acceptable solution; 2) quality of the solution defined as the ratio of events lowered during an execution, which is correlated to speedup; and 3) the size of the input space at the beginning of tuning stage/process.

a) Tuning Based on Static Program Data (`SLOC`): When considering tuning based on `SLOC`, we are interested to understand the impact of the incremental filtering during the search. The choices are search based on individual `SLOC`, search based on `SLOC` clusters obtained through a clustering algorithm and search using incremental filtering using clustering followed by individual events. These choices are referred to as `SLOC`, `SLOC-C` and `SLOC-C` → `SLOC` respectively in Figures 12 and 13. The data indicates that `SLOC-C` ramps up faster than `SLOC`. `SLOC-C` succeeds in lowering 84.31% of the calls in PMF 1D, while `SLOC` succeeds in lowering 69.08% calls in the same number of steps. When composing `SLOC-C` with `SLOC`, the number of inputs to the `SLOC` stage is 5, compared to 7 when running `SLOC` by itself. As `SLOC-C` with `SLOC` takes 2 steps to finish, using `SLOC-C` with `SLOC` does not reduce the final number of steps to get to the best solution (steps 7 to 8 on figure 12). Nevertheless, it goes faster to a better solution, steps 0 to 1 on figure 12. Similar trends are observed for PMF 2D, figure 13.

Thus the first observation is that when searching based on `SLOC` *data, the incremental filtering alternative* `SLOC-C` → `SLOC` *is preferred.*

b) Tuning Based on Backtrace (Dynamic) Data (`BT`): Figures 15 and 16 show the evolution of the search using classification based on backtrace (`BT`) data. When comparing against `SLOC` for PMF 1D, `BT` strategies reach the same quality of the solution although in larger number of steps due to the larger search space. For PMF 2D, `BT` improves the quality of the solution to 78% from 70% with `SLOC`. Overall the data indicates that `BT` increases the number of calls lowered and should therefore be incorporated into the search. The data also indicates that incremental filtering is beneficial and `BT-C` → `BT` is the preferred choice.

c) Impact of Clustering Strategies: In Section IV we discuss two clustering methods: a generic community detection which can return multiple results based on its arguments and a deterministic procedure returning a single result. We note that community detection will eventually return the result of the deterministic algorithm depending on selecting its input parameter. When considering clustering of backtrace data (`BT`) we argue for an initial step of preclustering using static data based on `SLOC`.

Intuitively there's a trade-off between the granularity of the cluster and chance of success for the search. Larger clusters will lead to fewer search steps, but will also increase the chance of failure when tuning.

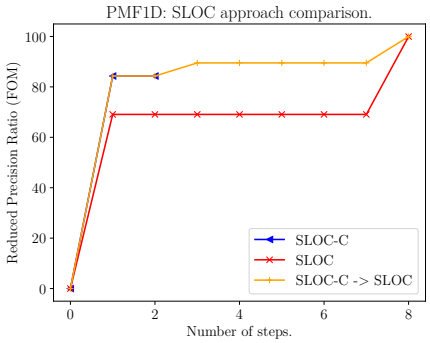

Fig. 12: *SLOC strategies on PMF 1D.*

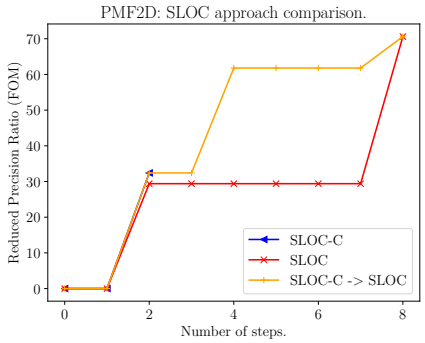

Fig. 13: *SLOC strategies on PMF 2D.*

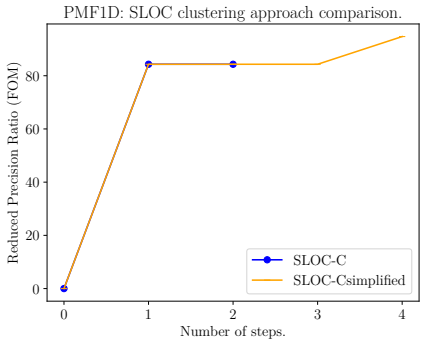

Fig. 14: *Clustering PMF 1D.*

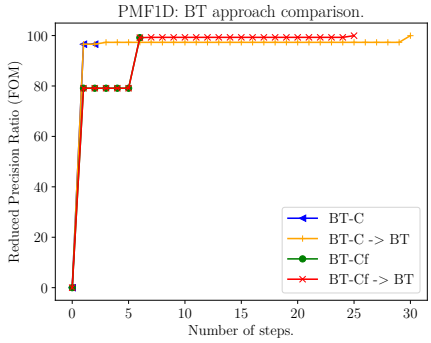

Fig. 15: *Backtrace strategies PMF 1D.*

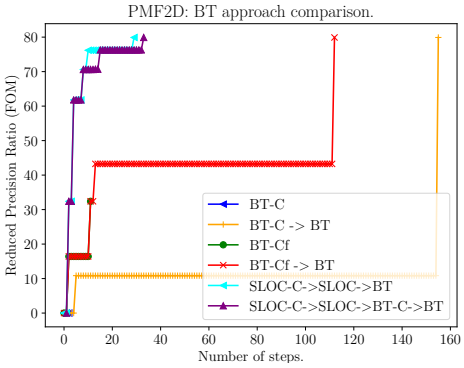

Fig. 16: *Comparison of Backtrace strategies on PMF 2D.*

In Figure 14 we present the evolution of the `SLOC-C` algorithm for PMF 1D on two inputs: `SLOC-C` is using community detection, `SLOC-Csimplified` is using our procedure. In the terms used in Section IV we compare the search over $\{(B,G),(R,P,Br,Gr,Pi)\}$ with $\{(B,G),(R,P),(Br,Gr,Pi)\}$. The data illustrates the trade-off between granularity and succes of the search, with `SLOC-Csimplified` leading to a better solution.

The impact of preclustering backtrace data using `SLOC` clustering is illustrated in Figures 15 and 16. `BT-C` refers to clustering of data with no initial filtering, while `BT-Cf` refers to the procedure described in Section IV-B. For PMF 1D, `BT-C` reaches a better solution faster than `BT-Cf`. On the other hand, for PMF 2D `BT-C` fails to find any solution, while `BT-Cf` finds a decent solution. This is explained by above mentioned trade-off. The preclustering step introduced on `SLOC` criteria reduces the granularity of the backtrace clusters. The algorithm may take more steps but it is likelier to succeed. This is also substantiated when examining the data for the composition of incremental filtering using `SLOC-C` → `SLOC` → `BT` with the `SLOC-C`→`SLOC`→`BT-C`→`BT`. Both lower a similar number of calls (78%), with the former reaching the solution slightly faster.

The data indicates that using static information for preclustering of backtrace/dynamic information data is beneficial.

d) Composing a Search Algorithm: When examining the choice of a clustering algorithm, we prefer our deterministic procedure, but note that generic community detection is a perfectly valid choice due to its tunability. When considering clustering of dynamic data, preclustering using static information is desired, e.g.`BT-Cf`.

When examining the structure of the search algorithm we note incremental filtering is definitely beneficial and should be used. The first stage is best performed using static information as `SLOC-C` → `SLOC`. Considering dynamic behavior is also useful, and it should employed after the first step. Thus a complete algorithm can perform `SLOC-C` → `SLOC` → `BT` or `SLOC-C` → `SLOC` → `BT-Cf` → `BT`. In our experiments they both reach the same solution, with the former using slighlty fewer steps. The proper composition is application dependent, so for generality we recommend the default `SLOC-C` → `SLOC` → `BT-Cf` → `BT`.

IX. Discussion

This work showcases both the value and the challenges of performing floating point performance tuning in whole application settings. The benchmarks we examined are part of large scale DOE code modernization efforts: PeleC is part of the DOE Exascale Computing Project [7], while CCTBX is one of the NERSC [25] application readiness benchmarks. Both are required to exploit the potential of specialized reduced precision functional units in existing

and upcoming generations of GPUs. We report speedup and suggested changes for the original application source code; it is also easy to see the extensibility to any other function than `exp`. Thus our findings are of interest to any other application developer participating in same or similar programs.

a) To the needs of the many: We expect our approach to provide performance benefits for any other code or situations where the performance difference between multiple implementations of a function is not offset by the cost of converting data between formats at the call boundary. This is trivially satisfied when replacing high arithmetic intensity operations such as `exp` or `dgemm` on sufficiently large matrices. For codes with more stringent characteristics, techniques to amortize the conversion overhead may be required, leading to a need for different search strategy.

b) Pedal to the metal: Our approach tunes precision based on a classification that tries to preserve and uncover structure already present in iterative solvers, thus we try to manipulate large granularity "objects". To assess the overall quality of the solution, we performed an experiment where, after applying our method, we attempt to lower the remainder of the calls considering them on an individual basis. For PMF2D, this amounts to 15,000,000 events accounting for 20% of the trace. Running generic greedy search algorithms (binary search, incremental sweep) as introduced in [28], [2] failed due to the sheer volume of the search space and a non-obvious structure of the solution. After careful examination of the source code, we were able to devise a strategy able to lower 18% of the remaining 20% of the calls. For reference, the strategy aligns blocks of 21 events in the trace, and keeps in double only one element in a fixed position within all blocks. This was deduced after source code inspection and we could not devise a proper refactoring to improve performance. Besides the obvious scalability challenge, we consider refactoring as a likely concern for other codes when considering events at this very fine granularity. Nevertheless, the results indicate that our approach is not necessarily optimal.

c) Precision error propagation can be unpredictable: Program behavior can change during precision tuning. In PMF2D, when modifying the precision of calls associated with some backtraces (`comp_Kc` in `LiDryer.cpp`), we observe control flow divergence and callstacks for `exp` not encountered during the profiling phase. We conservatively execute newly encountered `exp` invocations in double precision.

d) End Application Correctness Criteria Is Hard: One of the challenges of this work was obtaining the acceptability criteria from the application developers. This spurred a long conversation and iterative dialogue where they tried to develop criteria from first principles or empirically. The outcome of this exercise was an appreciation for tools that enable "easy" experimentation with different precisions, as this is a prerequisite for code refactoring for hardware specialization. A corollary is that compiler or toolchain specific tools are likely considered insufficient to address the needs of full multi-language, multi-library, multi-solver applications.

e) Teach Them How to Fish: We present a discussion on how to classify, filter program data and guide searches along multiple orthogonal criteria (SLOC, backtrace), (cluster,individual events) or (CFG, temporal). While we can provide a decent default structure for the search algorithm using incremental filtering based on these criteria, the optimal structure seems to be application dependent. Indeed, clustering increases search item granularity and leads to faster searches (fewer inputs). On the other hand, higher granularity increases the chance of failure, leading to an increased number of trials downstream. Thus, when targeting full applications, tuning tools are better off providing composable building blocks with guidelines, rather than a prescribed monolithic algorithm.

f) There's Room for Everybody: We can view our tool as a meta-search that does not displace any of the existing solutions. For example, the process of tuning an application to new hardware always starts with identifying a few kernels. These kernels are small and can be probably tuned by specializing our approach, but they are probably best transformed with an automated compiler approach. After producing multiple kernel versions with different precision behavior, our approach can be used for tuning the whole application to select the appropriate kernel. In this case the compiler based tool may face challenges.

X. Related Work

The prior sections of this paper have discussed multiple precision tuning tools [28], [18], [17], [15], [2] in order to compare and contrast them with the methods we propose here. These techniques vary in terms of how they characterize the search space, hierarchically represent event, and employ different search strategies; as such, they vary in terms of optimality, scalability and robustness to perturbations in the initial order of events.

a) Pruning Based on Floating-point Error Analysis: Another strategy, which we have not yet discussed, is to prune searches based on the error of the events that we are trying to tune. Adapt [23] employs algorithmic differentiation [24], which has scalability challenges due to its computational intensity. GPUMixer [15] employs a heuristic that sorts the list of data type configurations according to their accumulated relative error profiling; while this approach seems promising, tools such as Herbgrind [30] have demonstrated that high local relative error does not necessarily imply the existence of a high impact on the accuracy of a tuned program. To improve the quality of the solution GPUMixer employs an offline analysis step that combines information from multiple traces to suggest better candidates for precision tuning. The search proceeds then in a manner similar to ours.

Herbgrind [30] is one among a larger set of tools that aim to reason about sources of error in floating-point

applications. While the vast majority of these tools have not been applied to precision tuning, there is no principle reason to believe that they could not be used. While the paragraphs above focused on search space pruning, floating-point error analysis could also be incorporated into preprocessing steps, such as cluster formation.

FLIT [31], for example, allows the user to better understand the impact of compiler optimization flags (e.g. INTEL `-fp-model fast=1`). Shaman [3] is a C++11 library that applies operator overloading to evaluate the numerical accuracy of an application. Verrou [8] and Verificarlo [4], [1], [2] analyze floating-point error using Monte Carlo Arithmetic (MCA), while CESTAC [32] and CADNA [13] use interval arithmetic analysis to track floating point error. Lam et al. develop a binary instrumentation based tool to detect floating-point cancellation [19], while its counterpart, FPChecker detects floating-Point exceptions in GPU applications [14]. As mentioned above, future work could benefit from incorporating the results of these tools into precision tuning.

b) Precision Tuning Granularity and Hierarchy: A second category of papers tune precision by first selecting the elements to tune, followed by a tuning method, which may either be top-down or bottom-up, and can vary its granularity at each level of the search.

Our approach exclusively tunes transcendental math function calls, coarse-grained static call sites, and fine-grained dynamic calls. Our coarse-grained elements are a subset of what several prior techniques tune [28], [29], [11], but our work is unique in the use of the dynamic temporal component as a mechanism to incorporate fine-grained analysis into our approach.

XI. Conclusion

We present a methodology for tuning the precision of full fledged scientific applications written using multiple programming languages: Python, C++, CUDA and Fortran. We discuss the basic building blocks involved in building a precision tuner and their trade-offs. The novelty of our approach comes from a multi criteria classification of the application behavior using a combination of static and dynamic program information. Adding dynamic program information (backtrace classification) increases the opportunity for optimization by slicing in the temporal direction of the program's behavior. The multi-criteria classification enables building scalable search algorithms using an incremental refinement approach.

XII. Aknowledgements

This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] Yohan Chatelain, Pablo de Oliveira Castro, Eric Petit, David Defour, Jordan Bieder, and Marc Torrent. Veritracer: Context-enriched tracer for floating-point arithmetic analysis. In *25th IEEE Symposium on Computer Arithmetic, ARITH 2018, Amherst, MA, USA, June 25-27, 2018*, pages 61–68. IEEE, 2018.

[2] Yohan Chatelain, Eric Petit, Pablo de Oliveira Castro, Ghislain Lartigue, and David Defour. Automatic exploration of reduced floating-point representations in iterative methods. In Ramin Yahyapour, editor, *Euro-Par 2019: Parallel Processing - 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26-30, 2019, Proceedings*, volume 11725 of *Lecture Notes in Computer Science*, pages 481–494. Springer, 2019.

[3] Nestor DEMEURE. Shaman: Evaluate the numerical accuracy of an application. https://gitlab.com/numerical_shaman/shaman, 2020. [Online; accessed 15-February-2020].

[4] Christophe Denis, Pablo de Oliveira Castro, and Eric Petit. Verificarlo: Checking floating point accuracy through monte carlo arithmetic. In Paolo Montuschi, Michael J. Schulte, Javier Hormigo, Stuart F. Oberman, and Nathalie Revol, editors, *23nd IEEE Symposium on Computer Arithmetic, ARITH 2016, Silicon Valley, CA, USA, July 10-13, 2016*, pages 55–62. IEEE Computer Society, 2016.

[5] CCTBX developers. CCTBX framework. https://cctbx.github.io, 2020. [Online; accessed 01-January-2020].

[6] PeleC developers. presentation website. https://github.com/AMReX-Combustion/PeleC, 2017. [Online; accessed 01-January-2020].

[7] DOE. ECP. https://www.exascaleproject.org, 2020. [Online; accessed 22-April-2020].

[8] François Févotte and Bruno Lathuilière. Debugging and optimization of HPC programs with the verrou tool. In Ignacio Laguna and Cindy Rubio-González, editors, *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness), Denver, CO, USA, November 18, 2019*, pages 1–10. IEEE, 2019.

[9] Santo Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):75 – 174, 2010.

[10] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. 99(12):7821–7826, 2002.

[11] Hui Guo and Cindy Rubio-González. Exploiting community structure for floating-point precision tuning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 333–343, New York, NY, USA, 2018. Association for Computing Machinery.

[12] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J. Higham. Harnessing gpu tensor cores for fast fp16 arithmetic to speed up mixed-precision iterative refinement solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press, 2018.

[13] Fabienne Jezequel and Jean Chesneaux. Cadna: a library for estimating round-off error propagation. *Computer Physics Communications*, 178:933–955, 06 2008.

[14] I. Laguna. Fpchecker: Detecting floating-point exceptions in gpu applications. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1126–1129, Nov 2019.

[15] Ignacio Laguna, P. C. Wood, Ranvijay Pratap Singh, and Saurabh Bagchi. Gpumixer: Performance-driven floating-point tuning for gpu scientific applications. In *ISC*, 2019.

[16] 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)*, pages 27–35, Nov 2019.

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

[18] Michael O. Lam, Jeffrey K. Hollingsworth, Bronis R. de Supinski, and Matthew P. Legendre. Automatically adapting programs for mixed-precision floating-point computation. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, page 369–378, New York, NY, USA, 2013. Association for Computing Machinery.

[19] Michael O. Lam, Jeffrey K. Hollingsworth, and G.W. Stewart. Dynamic floating-point cancellation detection. *Parallel Computing*, 39(3):146 – 155, 2013. High-performance Infrastructure for Scalable Tools.

[20] Dorothee Liebschner, Pavel V. Afonine, Matthew L. Baker, Gábor Bunkóczi, Vincent B. Chen, Tristan I. Croll, Bradley Hintze, Li-Wei Hung, Swati Jain, Airlie J. McCoy, Nigel W. Moriarty, Robert D. Oeffner, Billy K. Poon, Michael G. Prisant, Randy J. Read, Jane S. Richardson, David C. Richardson, Massimo D. Sammito, Oleg V. Sobolev, Duncan H. Stockwell, Thomas C. Terwilliger, Alexandre G. Urzhumtsev, Lizbeth L. Videau, Christopher J. Williams, and Paul D. Adams. Macromolecular structure determination using X-rays, neutrons and electrons: recent developments in *Phenix*. *Acta Crystallographica Section D*, 75(10):861–877, Oct 2019.

[21] Zhenqi Lu, Johan Wahlström, and Arye Nehorai. Community detection in complex networks via clique conductance. *Nature - Scientific Reports*, 8(1):5982, 2018.

[22] Derek Mendez and Billy K. Poon. CUDA kernel. https://github.com/cctbx/cctbx_project/blob/master/simtbx/nanoBragg/nanoBraggCUDA.cu, 2020. [Online; accessed 01-January-2020].

[23] Harshitha Menon, Michael O. Lam, Daniel Osei-Kuffuor, Markus Schordan, Scott Lloyd, Kathryn Mohror, and Jeffrey Hittinger. Adapt: Algorithmic differentiation applied to floating-point precision tuning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press, 2018.

[24] Uwe Naumann. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*. Society for Industrial and Applied Mathematics, USA, 2012.

[25] NERSC. NESAP. https://www.nersc.gov/research-and-development/nesap/, 2020. [Online; accessed 22-April-2020].

[26] David Poliakoff and Matt LeGendre. Gotcha: An function-wrapping interface for hpc tools. In *ESPT/VPA@SC*, 2017.

[27] Pele project. presentation website. https://amrex-combustion.github.io, 2017. [Online; accessed 01-January-2020].

[28] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[29] C. Rubio-González, C. Nguyen, B. Mehne, K. Sen, J. Demmel, W. Kahan, C. Iancu, W. Lavrijsen, D. H. Bailey, and D. Hough. Floating-point precision tuning using blame analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1074–1085, May 2016.

[30] Alex Sanchez-Stern, Pavel Panchekha, Sorin Lerner, and Zachary Tatlock. Finding root causes of floating point error with herbgrind. *CoRR*, abs/1705.10416, 2017.

[31] Geoffrey Sawaya, Michael Bentley, Ian Briggs, Ganesh Gopalakrishnan, and Dong H. Ahn. Flit: Cross-platform floating-point result-consistency tester and workload. In *2017 IEEE International Symposium on Workload Characterization, IISWC 2017, Seattle, WA, USA, October 1-3, 2017*, pages 229–238. IEEE Computer Society, 2017.

[32] Jean Vignes. Discrete stochastic arithmetic for validating results of numerical software. *Numerical Algorithms*, 37(1):377–390, Dec 2004.

[33] Zhao Yang, René Algesheimer, and Claudio J. Tessone. A comparative analysis of community detection algorithms on artificial networks. *Scientific Reports*, 6(1):30750, 2016.

[34] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.