HPC-MixPBench: An HPC Benchmark Suite for Mixed-Precision Analysis

Konstantinos Parasyris*, Ignacio Laguna*, Harshitha Menon*, Markus Schordan*, Daniel Osei-Kuffuor*, Giorgis Georgakoudis*, Michael O. Lam† and Tristan Vanderbruggen*
* Lawrence Livermore National Laboratory
{parasyris1, lagunaperalt1, harshitha, schordan1, oseikuffuor1, georgakoudis1, vanderbrugge1}@llnl.gov
† James Madison University
lam2mo@jmu.edu

***Abstract*—With the increasing interest in applying approximate computing to HPC applications, representative benchmarks are needed to evaluate and compare various approximate computing algorithms and programming frameworks. To this end, we propose HPC-MixPBench, a benchmark suite consisting of a representative set of kernels and benchmarks that are widely used in HPC domain. HPC-MixPBench has a test harness framework where different tools can be plugged in and evaluated on the set of benchmarks. We demonstrate the effectiveness of our benchmark suite by evaluating several mixed-precision algorithms implemented in FloatSmith, a tool for floating-point mixed-precision approximation analysis. We report several insights about the mixed-precision algorithms that we compare, which we expect can help users of these methods choose the right method for their workload. We envision that this benchmark suite will evolve into a standard set of HPC benchmarks for comparing different approximate computing techniques.**

I. Introduction

As HPC programmers increasingly look for beyond-Moore computing techniques to increase computational throughput, *approximate computing* is emerging as a promising method to increase peak performance. Approximate computing is based on the observation that although performing exact computations at scale requires a high amount of computational resources, allowing certain approximations or occasional violations of numerical consistency can provide significant gains in efficiency [1], [2]. Approximate computing exploits the gap between the level of accuracy required by the applications and that provided by the system and has the potential to benefit a wide range of applications, including data analytics, scientific computing, machine learning, and many others.

Among the different approximate computing methods that currently exist, floating-point precision reduction, or *mixed-precision*, has recently become a popular approach. Modern-day computer architectures support multiple levels of precision for floating-point computations to provide tradeoffs between accuracy and performance. Several recent studies have demonstrated the use of mixed precision (i.e., using multiple levels of precision in a single program) to increase significantly the performance of scientific applications [3], [4], [5]. With accelerators supporting several levels of floating-point precision (e.g., half, single, and double precision in NVIDIA GPUs and others) and with higher peak performance in lower precision in these accelerators, this technique has become a promising approach to boost performance. As HPC systems are becoming increasingly heterogeneous, we expect mixed-precision to become more prevalent in scientific applications.

While a significant number of tools and methods have been proposed to tune mixed-precision applications, these tools and methods are often evaluated on very different workloads or benchmarks. Several tools are evaluated on a large set of very small programs, e.g., small math functions, while other tools are only demonstrated on a small set of realistic applications. This large discrepancy in the evaluation benchmarks that are used in all previous studies precludes the community from having a broad picture of the effectiveness of mixed-precision tools. Having a comprehensive set of benchmarks to consistently evaluate mixed-precision techniques could undoubtedly benefit developers and users of these methods.

Our Contributions: In this paper, we present HPC-MixPBench, a benchmark suite of programs for mixed-precision computing analysis. Our set of benchmarks is composed of 10 kernel codes and 7 application codes that represent common HPC workloads, some of which are used in the procurement of HPC systems. To evaluate the capability of the benchmark suite, we demonstrate them on a number of mixed-precision algorithms that have been proposed in the literature [6], [7], [8], and report several insights previously unreported about these mixed-precision algorithms. The findings we report in the paper can help programmers choose the appropriate mixed-precision method for their application or workload; reporting such insights was not possible before because of the absence of such a complete benchmark suite.

More specifically, we implement and compare 6 search algorithms for mixed-precision tuning: combinational [8], compositional [8], delta-debugging [7], hierarchical [6], hierarchical-compositional [8] and a Genetic Search Algorithm (GA) [3]. Our study is the first in the literature to compare such a large set of mixed-precision algorithms.

Although in some instances open-source tools that implement such algorithms exist, in some other cases the code of such methods is not publicly available. To perform a fair comparison of the methods, we implement them all in the FloatSmith framework [8]. FloatSmith facilitates the integra-

978-1-7281-7645-1/20/$31.00 ©2020 IEEE
DOI 10.1109/IISWC50251.2020.00012

tion of tools by providing a JSON-based interchange format and exposes the CRAFT [6], [9] generic search tool that allows for easy implementation of new search strategies and methods.

In summary, our contributions are:

1) We present the first benchmark suite for mixed-precision analysis. We enable a more comprehensive analysis of mixed-precision tuning techniques, which have become popular recently in boosting performance by selectively reducing floating-point precision.
2) We describe the process of choosing and building the benchmark suite to cover several HPC workloads. We focus on establishing benchmarks that represent important workloads in the domain of HPC, in contrast to previous work, which characterize workloads of other domains, such as multimedia or signal processing.
3) We demonstrate the benchmarks on the most used mixed-precision algorithms that have been reported in the literature. To capture all algorithms, we extend CRAFT with a new search strategy based on a genetic algorithm. The algorithm starts with a random set of lower precision variables and converges by making permutations to the most efficient configurations. Our evaluation reveals several insights about the advantages and disadvantages of the existing mixed-precision algorithms, which have not been previously reported. For example, the delta debugging algorithm typically results in configurations providing the most speedup. We expect our findings will guide programmers in choosing appropriate mixed-precision algorithms for their workloads.

II. Background

Mixed-precision analysis tools identify application transformation for which the application executes faster while preserving an acceptable level of accuracy. The speedup in execution time is obtained by transforming application parts from a higher floating-point precision to lower precision in order to take advantage of optimizations like instruction vectorization and to reduce the memory footprint.

Program Locations. Typically, mixed-precision tools apply search algorithms to identify such application transformations. The algorithm operates on a search space that consists of all possible application locations (loc) where the precision can be changed. For a source-code-based analysis, these locations can be variables, function parameters, or pointers. For a lower-level analysis (for example on LLVM IR), the locations can be any SSA register that represents a real number. Each of these locations could be transformed to use up to p precision levels, where p denotes the number of available precision types (e.g., $p = 3$ for an architecture that supports half, single, and double precision). The size of the search space is exponential (p^{loc}), preventing an exhaustive search in most cases.

Search Granularity. The choice of search algorithm granularity (e.g., variables vs. LLVM IR) involves various tradeoffs. For example, searching at a lower level will expose more locations but the search space will be larger, which may result in longer analysis times. Searching at a higher level reduces the number of locations, reducing the search space but exposing fewer opportunities to use lower precision. In addition, at the source code level, transformations need to produce a valid source code file (i.e., one that compiles without errors). Changing the type of a single variable may require a change to other variables or function parameters, particularly when arrays and pointer types are considered. To automatically detect which variables are semantically connected and require to have the same type we use *Typeforge*[1] (II-C) a tool that performs source code analysis and identifies a set of variables (a *cluster*) that must have the same type. This restriction further reduces the search space of the source code level mixed-precision techniques.

Our work currently focuses on source-level mixed-precision techniques that use a clustering algorithm to identify such variable clusters. We also currently focus on two precision levels: *double* (64 bits) and *single* (32 bits).

A. Mixed-Precision Automation

We use FloatSmith [8], a framework that integrates several existing tools to provide automated source-level mixed-precision search and transformation. We investigated other search based mixed-precision tools, namely Precimonious [7] and HiFPTuner [10]; however, those tools do not provide support for changing the type of dynamic memory allocation variables, which means they have limited applicability for HPC workloads. FloatSmith provides a generic framework to implement search strategies for mixed-precision tuning. We use it to evaluate existing strategies and to implement new ones.

For a given program, FloatSmith requires the user to provide instructions on how to acquire, build, and run the program as well as how to verify the output for correctness. Given this information, it uses TypeForge to extract a list of variables that can be converted from double precision to single precision. Then, it performs a search using CRAFT to determine which variables can be replaced (using transformations enabled by TypeForge) to achieve a speedup while still passing the user-specified verification routine.

B. Search Algorithms

We use CRAFT [6], [9] to search for valid mixed-precision configurations that pass the verification routine and exhibit a speedup over the original program. CRAFT has several built-in search strategies, and we also added a new strategy based on a genetic algorithm. We evaluate the following search strategies:

Combinational [8]: Try all combinations of variables or clusters: the brute-force or exhaustive search approach.

Compositional [8]: Replace each variable or cluster individually, then repeatedly combine passing configurations. Heuristics are used to reduce the number of configurations, but this strategy will be as slow as the combinational strategy when many variables can be replaced. The search terminates when there are no compositions left.

[1]https://github.com/LLNL/typeforge

Delta debugging [7], [10], [5], [8]: Use a modified binary search on the list of program variables or clusters. It terminates when it has reached a local minimum in which it cannot convert any more variables.

Hierarchical [6]: Use program structure information (e.g., modules or functions) to search for larger groups of variables that can be replaced, falling back to lower-level components and eventually to individual variables if necessary. This search does not incorporate cluster information because clusters may cross the function or module boundaries.

Hierarchical-compositional [8]: Integrates the hierarchical and compositional approaches, using the former to identify program components amenable to replacement and then using the latter to combine these individual components to search for speedups by replacing multiple components. The goal is to find more inter-component mixed-precision configurations without having to start by trying every variable individually. The search terminates when all passing configurations have been composed of other passing configurations.

Genetic Algorithm [11]: Mimics the process of natural selection. GA starts with a population of random configurations, where a configuration is an array of bits that represents the precision of the program variables. The evolutionary algorithm then proceeds iteratively, selecting individuals from each iteration or generation that are the best fit to produce offspring. In our algorithm, the fittest individual is the one that gives the best performance while satisfying the error criteria. Afterward, the GA algorithm combines the configurations of best fit individuals and randomly mutates them to create the next generation. The algorithm terminates when a maximum number of generations have been created or when the best-fit individual of the population doesn't change for several iterations.

C. Type Dependence Analysis and Type Refactoring

Typeforge enables Floatsmith to perform type refactoring at the source level. Typeforge can change the type of variable declarations, function parameters, function return-types, and template instantiation type-arguments (for both functions and classes). To ensure that refactored programs always compile, Typeforge performs an inter-procedural type-dependence analysis to determine sets of type-changes that must be performed together. Similar to [12], for a given type-correct program, an entity x is type-dependent on an entity y if and only if x's type may need to be changed as a consequence of a change in y's type to maintain type correctness. In contrast to [12], Typeforge's analysis is purely based on type information and does not perform an over-approximating value-flow analysis. This guarantees that we can compute a partitioning of the set of all relevant type-changes, i.e. the type-change sets are disjunctive. Each type-change set maps to one *cluster* of multiple variables (i.e., one that will compile) in Floatsmith's search algorithm.

We provide an example input code in Listing 1. This code demonstrates type dependencies. For example, the variable `arr` and function parameter `input` must have the same base type because `arr` is used in a call to the function `vect_mult` and both have pointer types. Similarly, a pointer to `val` is used as the argument for the parameter `inout` in this call, hence `val`'s type must be the same as the base type of `inout`. However, the variables scale and parameter ratio are not dependent as the value can be cast if one is changed.

```
void vect_mult(int n, double * input, double * inout,
   double ratio) {
  double res;
  for (int i = 0; i < n; i++) {
    res += ratio * input[i];
  }
  *inout += res;
}

void foo() {
  double arr[10]; init(10, arr);
  double val = init_scalar();
  double scale = init_scalar();
  vect_mult(10, arr, &val, scale);
}
```

Listing 1: **Type dependency example.** The variable `arr` and function parameter `input` must have the same base type, and so do `val` and `inout`. This information is used to create partitions that are then used to create *clusters* of multiple variables for search algorithms.

For the above example, the following partitioning of variables is computed: {`arr`, `input`}, {`val`, `inout`}, {`scale`}, {`ratio`}, and {`res`}. Variables in a set must have the same base type for the program to compile, and so the power set describes the set of all possible changes to derive a valid configuration of the program. The goal of a mixed-precision search is to find a set (or multiple sets) of variables to replace that will improve the performance. Changing all variables to single precision might result to a faster program, but changing {`arr`, `input`} and {`ratio`} might also provide a speedup, whereas changing {`arr`, `input`} alone will only add additional casts and make the program slower.

III. Benchmark Suite and Evaluation Framework

HPC-MixPBench has two design goals: (1) to provide a set of kernels and proxy applications that are representative of HPC workloads along with a set of approximating techniques applicable to them, and (2) to provide extensible interfaces for integrating new approximation techniques, including interfaces for analyzing their performance and correctness.

Typically, deploying and analyzing the performance of an approximation technique involves several steps, including instrumentation, profiling, and verification. HPC-MixPBench simplifies the specification of those steps, helping the user to focus on evaluating approximation techniques rather than spending time on implementation details.

A. Overview

Figure 1 presents an overview of HPC-MixPBench. Specifically, HPC-MixPBench consists of a set of benchmark applications, a runtime library for instrumentation and profiling, and a verification library to evaluate the accuracy loss when

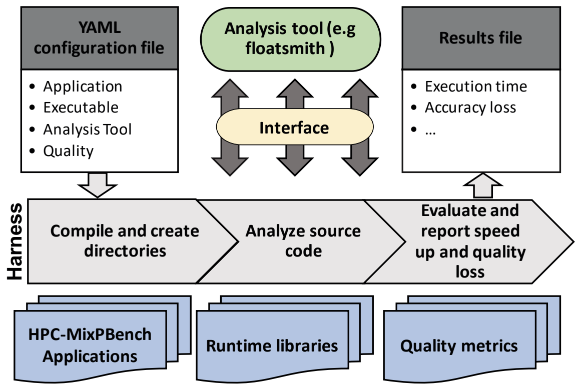


Fig. 1: **High level overview of the HPC-MixPBench framework.** HPC-MixPBench includes a set of benchmark applications, a runtime library for profiling, and a verification library to evaluate the specified quality metric. HPC-MixPBench has a harness to execute the benchmarks based on the information provided in the YAML configuration file.

```
void foo(double **ptr, int elements){
    FILE *fd = fopen("input.bin", "rb");
    size_t allocationSize = sizeof(double)*elements;
    *ptr = (double*) malloc(allocationSize);
    fread( *ptr,sizeof(double), elements, fd);
    fclose(fd);
    performComputation(*ptr, elements);
    fd = fopen("output.bin", "wb");
    fwrite(*ptr, sizeof(double), elements, fd);
    fclose(fd);
    return;
}
```

Listing 2: **Example code without HPC-MixPBench library support**, any modification using a static analysis tool of the pointers types, would result into erroneous behavior.

approximating an application. Also, HPC-MixPBench provides a *harness framework* to deploy and evaluate benchmark applications. Next, we provide more details on the individual components comprising HPC-MixPBench.

a) Runtime library: Often mixed-precision static analysis tools that operate on the source code change the type of variables and pointers. However, these tools are unable to change the interaction of the application with input files or the way memory is allocated. An example of such a hard to analyze case is demonstrated in Listing 2. The user opens a binary file (line 2), allocates memory (line 4), reads the file (line 5), performs computations (line 7), and stores the result into a binary file (line 9). Converting this code to enable mixed-precision execution is non-trivial, because both the memory allocation and file reading/writing must be made compatible to execute correctly with either single or double precision, depending on the configuration dictated by the applied approximation technique.

All these technical issues are typically handled with custom solutions. HPC-MixPBench provides a runtime API for this. The library provides variants of C library functions that are necessary to support mixed-precision execution, including memory allocation and file I/O. In Listing 3 we present the source code after applying the HPC-MixPBench API. The *fread, malloc, fwrite* operations are replaced by *mp_fread, mp_malloc, mp_fwrite* calls. In the case of I/O calls, the user describes the files based on their initial type (in the provided example the initial type is *DOUBLE*). Should a mixed-precision alter the type of *ptr* from double to single precision, the *mp_fread, mp_fwrite* calls will handle any necessary data conversions. These changes are already applied to all applications included in HPC-MixPBench.

b) Verification library: The verification library facilitates the comparison of the output of the original, non-approximate application to its approximated version. For this comparison, it provides several error metrics to quantify accuracy loss, including Mean Absolute Error (MAE), Root Mean Square Error (RMSE), Mean Square Error (MSE), coefficient of determination (R^2), and Misclassification Rate (MCR). Depending on the application characteristics and the approximation technique of interest, different quality metrics might be preferred. For example, when large errors in continuous calculations need to be avoided *RMSE* should be preferred because it penalizes large errors more. However, from an interpretation standpoint, *MAE* is easier to understand and use. Finally, the verification library serves as a single point for providing verification extensions so that new metrics can be added.

```
void foo(double **ptr, int elements){
    FILE *fd = fopen("input.bin", "rb");
    *ptr = (double*) mp_malloc(elements,*ptr);
    mp_fread(*ptr,DOUBLE, elements, fd);
    fclose(fd);
    performComputation(*ptr, elements);
    fd = fopen("output.bin", "wb");
    mp_fwrite(*ptr, DOUBLE, elements, fd);
    fclose(fd);
    return;
}
```

Listing 3: **Example Code with HPC-MixPBench library support**. Any interaction with memory allocation and IO will be handler by our library.

c) Harness: The harness is implemented as a Python script that deploys and runs applications. It is guided by a user-provided YAML configuration file that describes how to execute and analyze the application. We provide configuration files for all existing benchmarks in HPC-MixPBench. Figure 4 shows the configuration file of the *K-means* benchmark[2] The file first provides instructions for building and cleaning the application. Then, the analysis clause provides an identifier for the analysis, the name of the class that implements the analysis, and any extra arguments necessary. The rest of the clauses provide more details about how to run the application and how to calculate the quality of a run.

The harness is extensible to implement different analysis techniques on a deployed application through a plugin interface. Implementing a new analysis technique entails extending

[2]For brevity we have removed some YAML syntax.

```
kmeans:
    build_dir: 'kmeans'
    build: ['make']
    clean: ['make clean']
    analysis:
        floatsmith:
            name: 'floatSmith'
            extra_args:
                algorithm: 'ddebug'
    output:
        option: '-o'
        name: 'outputFile.bin'
    metric: 'MAE'
    bin: 'kmeans'
    copy: ['kmeans', 'kdd_bin']
    args: '-i kdd_bin -k 5 -n 5'
```

Listing 4: **Example of YAML configuration file.** The configuration file describes the application and the analysis phase.

a base Python class, which defines an analysis function. The analysis function will be invoked by the harness when the analysis needs to take place. The function returns a path to the executable of the analyzed application. This interface allows HPC-MixPBench to facilitate the comparison of different analysis techniques and algorithms on the same set of applications.

The interface between the harness and the analysis tool is straightforward. The user provides to the harness the source directory of the application, a command which compiles the application, the executable name, the executable invocation command, and the quality estimation command. The harness returns a path to the executable produced by the analysis.

Invoking the harness with the YAML configuration file runs the analysis Python code, which compiles the application, executes the generated binaries, and performs the prescribed analysis and evaluation to quantify quality loss and to measure execution time under different approximation techniques.

B. Benchmark Descriptions

There are two types of benchmark programs in HPC-MixPBench: kernels, which are typical as building blocks of HPC codes, and larger HPC proxy or mini-applications.

The kernels are an ideal starting point to test a new approximate technique as they are easy to understand; usually, they do not use complex data structures but are still representative of modern HPC code snippets. Also, kernels do not perform any I/O and their input data are randomly initialized so they are easy to deploy and analyze. Table I lists the kernels included in HPC-MixPBench, along with short descriptions.

Further, HPC-MixPBench includes a set of HPC proxy and mini applications. We select applications from the [13], [14] HPC benchmarks suites that contain benchmarks representative of large HPC applications. We selected only applications that perform floating-point computations and we merged all source code files into a single file to allow easier automated analysis from other static analysis tools. HPC-MixPBench includes the following benchmarks:

Blackscholes: The application [13] calculates the prices for a portfolio of European stock options analytically by solving a Partial Differential Equation (PDE). As a quality metric we apply MAE between the stock prices.

TABLE I: **Kernels included in HPC-MixPBench**.

Name	Description
banded-lin-eq	Banded linear systems solution
diff-predictor	Difference predictor
eos	Equation of state fragment
gen-lin-recur	General linear recurrence equation
hydro-1d	Hydrodynamics fragment
iccg	Incomplete Cholesky conjugate gradient
innerprod	Inner product
int-predict	Integrade predictors
planckian	Planckian distribution
tridiag	Tridiagonal linear systems solution

CFD: The CFD [14] solver is an unstructured grid finite volume solver for the three-dimensional Euler equations applied to compressible flows. To quantify the accuracy loss of approximating CFD we apply the *MAE* quality metric to the density, momentum, and the energy density.

Hotspot: The HotSpot [14] application simulates heat dissipation to estimate processor temperature based on an architectural floor plan and simulated power measurements. The thermal simulation iteratively solves a series of differential equations. Each output cell in the computational grid represents the average temperature value of the corresponding area of the chip. The accuracy is quantified by comparing the final temperature of the floor plan across all grid points.

HPCCG: HPCCG[15] is a PDE application and preconditioned conjugate gradient solver that solves a linear system. The quality of the output is quantified by the residual of the solver.

K-means: K-means [14] is a clustering algorithm used in data-mining, important primarily for its simplicity. Many data-mining algorithms show a high degree of data parallelism. In K-means, a data object is comprised of several values, called features. By dividing a cluster of data objects into K sub-clusters, K-means represents all the data objects by the mean values or centroids of their respective sub-clusters. The algorithm associates each data object with its nearest center. The application outputs the assignment of objects into clusters. The accuracy is calculated using the *MCR* metric.

LavaMD: The code [14] calculates particle potential and relocation due to mutual forces between particles within a large 3D space. This space is divided into cubes, or large boxes, that are allocated to individual cluster nodes. The large box at each node is further divided into cubes, called boxes. 26 neighbor boxes surround each box (the home box). Home boxes at the boundaries of the particle space have fewer neighbors. Particles only interact with those other particles that are within a cutoff radius since particles at larger distances exert negligible forces. We apply the MAE metric on the location and the velocity of each particle.

SRAD Speckle Reducing Anisotropic Diffusion (SRAD) [14] is a diffusion method for ultrasonic and radar imaging applications based on partial differential

TABLE II: **Total Variables (TV) and Total Clusters (TC) identified by Typeforge as possible transformations.**

Kernels			Applications		
Name	**TV**	**TC**	**Name**	**TV**	**TC**
banded-lin-eq	2	1	Blackscholes	59	50
diff-predictor	5	1	CFD	195	25
eos	7	2	Hotspot	36	22
gen-lin-recur	4	1	HPCCG	54	27
hydro-1d	6	2	LavaMD	47	11
iccg	2	1	K-means	26	15
innerprod	3	2	SRAD	29	14
int-predict	9	2			
planckian	6	2			
tridiag	3	1			

equations (PDEs). It is used to remove locally correlated noise, known as speckles, without destroying important image features. We output the corrected image, and we apply the *MAE* metric on these images.

IV. Evaluation

In this section we showcase the ability of HPC-MixPBench to analyze several mixed-precision search algorithms described in section II-B and provide a holistic study on using them.

We use the following three metrics to evaluate the effectiveness of each algorithm:

Evaluated Configurations (**EV**) represents the number of configurations that the mixed-precision algorithm evaluated before reaching a solution. This metric is usually directly correlated with the total search time; higher numbers of tested configurations imply a longer search.

Speedup (**SU**) is the execution time speedup of the mixed-precision version discovered by the search algorithm in comparison to the execution time of the original program. We execute each version of the program ten times and take the average of the execution times after discarding the best and the worst. This is the most important metric since the analysis will always respect the quality constraint defined by the user. Higher speedups are better.

Accuracy (**AC**) indicates the error of the mixed-precision version, in comparison with the original execution. For simplicity, all applications use the *MAE* (Mean Absolute Error) quality metric except *K-Means* for which we use the *MCR* (misclassification rate) metric.

Each application was analyzed using all six search algorithms: *Combinational (CB), Compositional (CM), Delta-debugging (DD), Hierarchical (HR), Hierarchical-Compositional (HC)*, and *Genetic Algorithm (GA)*. Each search algorithm was given a fixed time limit to converge to a solution and the ones that did not find a solution were marked accordingly. We set the limit to twenty-four hours.

We use the support of HPC-MixPBench's harness to schedule each analysis in parallel on a cluster consisting of multiple nodes. Each node is equipped with an Intel 8-core Xeon E5-2670 and $256GB$ DRAM. The harness offloads the search for each combination of an application/algorithm to a separate node but executes all the final binaries on the same node for consistency. This is managed automatically by the harness.[3]

A. Benchmarks Analysis Complexity

In Table II we report the complexity of all applications. We provide two metrics, the Total number of Variables (TV) and the Total number of Clusters (TC). Some of the algorithms operate at the granularity of clusters (groups of variables), whereas others operate at the granularity of variables. The current implementations of *CM*, *HR*, and *HC* operate on variables, while the remaining algorithms operate on clusters.

As expected, the number of variables for the kernels is very small, and when the clustering is applied the search space is greatly reduced. Consequently, the kernel benchmarks serve as a good starting point for debugging purposes. Moreover, it allows us to apply the exhaustive search algorithm (CB) to find and compare against the best possible configuration for these kernels. Additionally, these kernels provide a standard set of benchmarks that can be used for evaluating algorithms and tools that may not scale well to larger benchmarks.

In contrast to the kernel benchmarks, the applications provide a larger scale for evaluating the search algorithms. While *CFD* has the largest number of variables (195), *Blackscholes* has the largest number of clusters (59). Interestingly, *Blackscholes* presents a small degree of variable clustering in comparison to other applications. In essence, the clustering algorithm presented in II-C enforces that assignments between pointer variables should always share the same type so these variables will be in the same cluster, whereas assignments between scalar variables do not impose the same restriction. Consequently, with *Blackscholes*, an application where most assignment operations are carried out between scalar variables, we observe that clustering does not significantly reduce the search space.

A closer look at the *CFD* program reveals that it operates on a limited amount of scalar variables, and most functions in the program use parameter array pointers. This structure of the program allows the clustering algorithm to group all these parameters into the same base type, thereby generating a small number of clusters. This indicates that *CFD* can take advantage of clustering to reduce the search space considerably.

B. Evaluation

1) Kernel Evaluation: In Table III we present the results when analyzing the kernels. We set the quality threshold to be 10^{-8}. Most algorithms converge to the same solution. Therefore in terms of quality most experiments present the same *MAE* value since they resulted in the same configuration. In the Quality sub-table, we mark with red cells the algorithm/kernel combinations that resulted in a different configuration than most of the algorithms. Both hierarchical approaches tend to produce such outputs, with larger quality degradation than the remaining algorithms and with smaller speedups. Thus, in 40% and 50% of the cases *HR* and *HC* converge to a suboptimal

[3]SLURM support for scheduling scripts is required

TABLE III: **Evaluation resultis of kernel codes**

Application	**Quality**(10^{-9})						**Evaluated Configs**						**Speedup**					
	CB	**CM**	**DD**	**HR**	**HC**	**GA**	**CB**	**CM**	**DD**	**HR**	**HC**	**GA**	**CB**	**CM**	**DD**	**HR**	**HC**	**GA**
banded-lin-eq	9.94	9.94	9.94	9.94	9.94	9.94	1	1	1	1	1	2	4.45	4.46	4.52	4.53	4.47	4.45
diff-predictor	9.94	9.94	9.94	9.94	9.94	9.94	1	1	1	1	1	2	1.6	1.6	1.6	1.6	1.6	1.6
eos	0.0	0.0	0.0	1.13	1.13	0.0	2	2	2	12	9	4	0.99	1.0	1.0	0.98	1.0	1.0
gen-lin-recur	0.0	0.0	0.0	6.39	6.39	0.0	1	1	1	7	6	2	0.98	1.01	1.01	0.92	0.91	1.0
hydro-1d	2.71	2.71	2.71	2.71	2.71	2.71	2	3	2	1	1	4	1.7	1.74	1.74	1.74	1.74	1.69
iccg	9.94	9.94	9.94	9.94	9.94	9.94	1	1	1	1	1	2	1.9	1.9	1.89	1.91	1.89	1.91
innerprod	0.0	0.0	0.0	0.0	0.0	0.0	2	2	2	5	5	4	1.01	1.01	1.01	1.01	1.01	1.01
int-predict	1.74	1.74	1.74	1.74	0.24	1.74	2	2	2	110	11	3	1.49	1.51	1.48	1.51	1.04	1.52
plankian	0.0	0.0	0.0	6.37	6.37	0.0	2	2	2	23	8	4	1.0	0.99	1.0	1.02	1.0	0.99
tridiag	0.0	0.0	0.0	6.42	6.42	0.0	1	1	1	8	5	2	0.99	1.0	0.99	1.02	1.01	1.0

solution. In the case of *int-predict*, the *HC* algorithm misses relaxation opportunities since the quality of output may be further improved without violating the user constraint. Moreover, we observe that *HR* and *HC* examine a much larger number of configurations for some benchmarks because they operate on individual variables instead of clusters.

Although these kernels do not provide any interesting insight on the behavior of the algorithms, they provide a simple and valuable set of applications that can be used to identify misbehaving cases during the implementation of algorithms in mixed-precision tools.

2) Application Evaluation: Table IV presents the maximum speedup of each application as well as the maximum accuracy loss each application can achieve. To determine these metrics, we manually changed all applications into their corresponding single precision versions and we compare the execution time and the quality with the original double-precision version. These measurements provide a nice insight into the limitations of mixed-precision search algorithms. The table contains two extreme cases. First, there is *K-means*, where even full single-precision conversion preserves the original quality without providing a significant performance benefit (for the input files we tested). On the other hand, *SRAD* yields a theoretical maximum speedup of 1.5x but the output quality is completely destroyed as the application outputs as some of the output computed values are NaN (Not-a-Number). *LavaMD* presents the highest potential in terms of speedup, with an accuracy loss of 10^{-4}. LavaMD accesses arrays with regular access patterns. When these arrays are converted into single precision, the cache miss rate of the application is reduced and the faster and simpler single precision instructions enable a significant speedup.

TABLE IV: **Application Speed Up and Quality loss** when comparing single- to double precision executions.

Application	**Speed Up**	**Quality Metric**	**Quality Loss**
Blackscholes	1.04	MAE	4.10E-06
CFD	1.38	MAE	1.10E-07
Hotspot	1.78	MAE	3.08E-10
HPCCG	1.00	MAE	2.0E-06
K-means	0.96	MCR	0
LavaMD	2.66	MAE	3.38E-04
SRAD	1.48	MAE	NaN

During our evaluation we set three different quality bounds: 10^{-3}, 10^{-6}, and 10^{-8}. In the most relaxed case of 10^{-3}, all applications (except *SRAD*) should be able to achieve speedups close to their single precision version. In the second case (10^{-6}) *LavaMD* should be amenable to a mixed-precision version. Finally, in the strictest quality bound 10^{-8}, only *K-means* and *Hotspot* should provide a pure single-precision version.

In Table V we present the results of our analysis. When the quality threshold is equal to 10^{-3}, *Hotspot* should be able to achieve the maximum speedup. Although *DD*, *HR*, and *HC* are able to convert variables and arrays into single precision, *Typeforge* does not handle literals. Therefore, since the literals are evaluated as double-precision the application is executing some extra typecasts which reduce the potential speedup. Moreover, *GA* does not identify the optimal configuration. In our setting, we significantly decrease the search time of *GA* by providing a small number of maximum iterations, and so the randomness of the algorithm prevents it from identifying configurations with speedups.

The *DD*, *HR*, and *HC* approaches terminate immediately due to their initial criteria. The algorithms applied changes on the coarsest hierarchy (effectively the entire application) and these changes produced a configuration that satisfies the threshold. This observation applies to all applications except *SRAD*. Finally, *CM* did not manage to terminate on multiple applications because it could not test the large number of configurations required within the time limit.

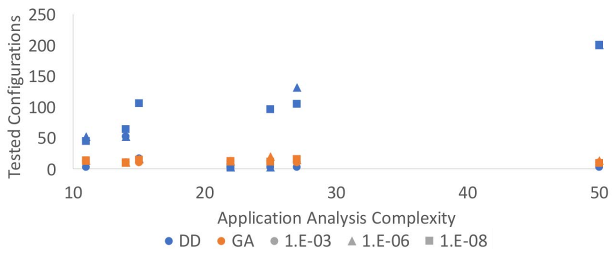
When setting the threshold to 10^{-6} the *HR* algorithm fails to identify a configuration for *Blackscholes*. Initially, it transformed the entire application into single precision, but this configuration fails the quality threshold. During the

TABLE V: **Evaluation results of the applications included in HPC-MixPBench**, while setting the quality threshold to $10^{-3}, 10^{-6}, 10^{-8}$ (empty gray boxes correspond to algorithms which did not produce any results in 24 hours).

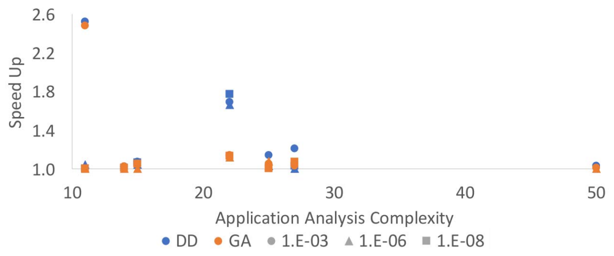
Application	**Speedup**					**Evaluated Configs**					**Quality (Threshold 10^{-3})**				
	CM	**DD**	**HR**	**HC**	**GA**	**CM**	**DD**	**HR**	**HC**	**GA**	**CM**	**DD**	**HR**	**HC**	**GA**
Blackscholes		1.03	1.01	1.02	1.01		2	2	2	10		10^{-6}	10^{-6}	10^{-6}	0
CFD		1.14	1.11	1.12	1.05		2	2	2	12		10^{-7}	10^{-7}	10^{-7}	10^{-7}
Hotspot		1.69	1.70	1.58	1.14		2	2	2	12		10^{-10}	10^{-10}	10^{-10}	10^{-10}
HPCCG		1.21	1.19	1.22	1.03		2	2	2	10		10^{-6}	10^{-6}	10^{-6}	10^{-14}
K-means		1.07	1.08	1.08	1.05		16	2	2	10		0	0	0	0
LavaMD	2.44	2.52	2.54	2.58	2.48	2048	2	2	2	12	10^{-4}	10^{-4}	10^{-4}	10^{-4}	10^{-4}
SRAD	1.0	1.02	1.0	1.02	1.02	26	52	32	39	10	0	0	0	0	0

Application	**Speedup**					**Evaluated Configs**					**Quality (Threshold 10^{-6})**				
	CM	**DD**	**HR**	**HC**	**GA**	**CM**	**DD**	**HR**	**HC**	**GA**	**CM**	**DD**	**HR**	**HC**	**GA**
Blackscholes		0.99		0.99	1.0		200		63	13		10^{-13}		10^{-7}	0
CFD		1.03	1.1	1.08	1.08		2	2	2	20		10^{-7}	10^{-7}	10^{-7}	10^{-7}
Hotspot		1.66	1.63	1.68	1.12		2	2	2	12		10^{-10}	10^{-10}	10^{-10}	10^{-10}
HPCCG		1.00	1.0		0.98		131	58		14		0	0		0
K-means		1.04	1.06	1.05	1.0		16	2	2	15		0	0	0	0
LavaMD	1.03	1.04	1.56	1.54	1.0	259	52	52	602	13	0	10^{-7}	10^{-7}	10^{-7}	10^{-7}
SRAD	1.0	1.0	1.0	1.0	1.0	26	52	32	39	10	0	0	0	0	0

Application	**Speedup**					**Evaluated Configs**					**Quality (Threshold 10^{-8})**				
	CM	**DD**	**HR**	**HC**	**GA**	**CM**	**DD**	**HR**	**HC**	**GA**	**CM**	**DD**	**HR**	**HC**	**GA**
Blackscholes	0.99	0.99		1.0	0.99	4134	200		65	9	10^{-13}	0		0	0
CFD		0.95		0.98	1.00		96		200	11		10^{-09}		10^{-11}	0
Hotspot		1.77	1.73	1.64	1.13		2	2	2	12		10^{-10}	10^{-10}	10^{-10}	10^{-10}
HPCCG		1.03	1.06		1.07		104	57		15		0	0		10^{-15}
K-means		1.06	1.07	1.08	1.05		106	2	2	14		0	0	0	0
LavaMD	1.0	1.0	1.0	1.0	1.0	23	44	52	67	13	0	0	0	0	0
SRAD	1.01	1.01	0.98	1.01	1.01	26	64	32	39	10	0	0	0	0	0



(a) **Correlation between the application analysis complexity and the efficiency of the mixed-precision search algorithm for different quality thresholds, when using DD or GA.** For stricter quality thresholds *DD* checks more configurations.



(b) **Correlation between the application analysis complexity and the obtained speedup for different quality thresholds, when using DD or GA**. The extra effort of *DD* rarely results to more performant configurations.

Fig. 2: *DD* vs *GA* for different applications and different threshold configurations.

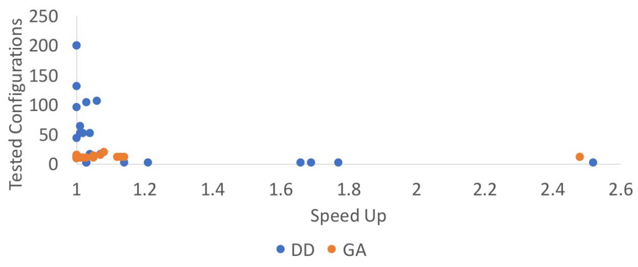


Fig. 3: **Correlation between the speedup of the application with number of tested configurations.**

ensuing search, it fails to find a valid configuration within the time limit. *HR* operates on the variable level and not on the cluster level, and consequently, the search space is much larger. Moreover, *HR* might select configurations that do not compile or raise run-time errors. Consequently, the algorithm wastes time on creating useless configurations.

In *Hotspot*, *DD*, *HR*, and *HC* find the same configuration as in the previous threshold setting, but *GA* converges to an entirely different configuration. This is due to the non-deterministic nature of genetic algorithms. When comparing the *DD* algorithm between the two thresholds, the algorithm requires more effort to converge into a solution because the number of evaluated configurations increased. For example, in *Blackscholes* the evaluated configurations increased from 2 to 200. Finally, when setting the quality threshold to 10^{-8} the limitations of *HR* become obvious as it fails to finish in 2 out of 7 applications.

Only *DD* and *GA* successfully searched and identified a valid mixed-precision configuration for all of our applications and all the different thresholds. In Figure 2a we compare the application complexity (x-axis), in terms of the total number of clusters, with the total tested configurations (y-axis). *DD* typically tests more configurations until it reaches a solution, whereas *GA* presents stable behavior by testing almost the same number of configurations. The cases where *DD* tests fewer configurations than *GA* are the cases where the search result is obvious (e.g., the entire application can be converted to single precision). Otherwise, *DD* requires more time.

In Figure 2b we compare the speedup, obtained by mixed-precision when using the two different algorithms (*GA* and *DD*). Both algorithms typically produce configurations with the same execution time. Typically, *DD* produces slightly more performant versions than *GA*. In Figure 3 we correlate the speedup of different search scenarios versus the amount of time required by the search algorithm to reach a solution. Most of the tested configurations resulted in a speedup between $1.0-1.2$. A limited number of scenarios were able to produce higher speedups.

V. Insights and Recommendations

Our evaluation revealed several insights about the mixed-precision search algorithms evaluated in this paper.

- From the perspective of tools and techniques for mixed-precision analysis, our work validated the need for creating viable mixed-precision configurations using cluster information. We show in our evaluation that applying mixed-precision search algorithms individually on variables, without considering whether they map on to a *valid configuration*, not only increases the search time but may also result in cases where the search algorithm fails to converge to a solution due to encountering *invalid configurations* along the search path.
- In LavaMD, we observe a large speed-up when reducing the precision of the application. When performing mixed-precision analysis on the application source code, lowering the precision of an array can change the cache behavior of the application, resulting in large speedups. Such opportunities cannot be discovered from tools that operate on the intermediate representation of the compiler [7], [16], [17] because they operate on a finer granularity and the application memory is not changed.
- The analysis time for GA is the easiest to predict among all search algorithms. The strict termination criterion of the algorithm (an upper bound on the number of iterations performed) bound the number of configurations being evaluated. However, the inherent randomness in the algorithm influences the determinism of the result.
- The search based on delta debugging typically results in configurations providing the most speedups. The algorithm can consistently exploit information obtained from the previous iterations and converge to a good solution. However, as the quality threshold gets stricter, the algorithm explores a large number of configurations resulting in high analysis time.
- Hierarchical approaches work well for relaxed thresholds; however, as the threshold becomes tighter, the algorithms require more steps to identify valid configurations that meet the error criteria as well as provide speedups. The current implementations of hierarchical approaches in CRAFT do not take into account clusters, as there is no straightforward way to use the clustering information without breaking the notion of hierarchy. However, the evaluation presented in this paper provides sufficient motivation to redesign these strategies to take clustering information into account to reduce the search space.
- Finally, we see in our evaluation that reducing the number of double precision variables does not always guarantee an improved execution time, so we do not recommend this as a viable search strategy. Other factors, such as compilation flags or operations on subnormal regions, might influence the execution time of the application. Consequently, auto-tuning by running the configuration as opposed to using a performance model or error helps identify such cases.

VI. Related Work

There is past work on autotuning approximate strategies. Precimonious [7] tunes floating-point precision to achieve performance improvement. A profiler for quality-of-service [18] evaluates the loop perforation technique by assessing their performance and quality impact. The ACCEPT framework [19] generalizes autotuning for several approximate computing techniques. However, these works do not provide a standard set of benchmarks for evaluating approximate computing. In contrast, our work aims to provide a comprehensive set of benchmarks to evaluate mixed-precision tools and techniques.

Our work is most similar to a recent effort to collect floating-point benchmarks called FPBench [20], [21]. FPBench focuses on static error analysis and formal mixed-precision guarantees, containing many smaller benchmarks that can be expressed as a single real-valued expression. In our work, we target larger benchmarks and proxy applications written in C++. Such benchmarks would be difficult or impossible to represent in the FPCore language used for FPBench. AxBench [22] is a benchmark suite for approximate computing consisting of a diverse set of applications in different domains such as machine learning and image processing, whereas our work aims to create a benchmark suite that represents common HPC workloads.

There is prior work in building mixed-precision configurations using software tools, including CRAFT [6], [9] and FloatSmith [8], which we leverage in this work to provide a basis for comparison between approaches. Other work in automated mixed-precision analysis includes Precimonious [7], [16], FPTuner [17], Daisy [23], [24], [25], HiFPTuner [10], ADAPT [4] GPUMixer [5], and AMPT-GA [11] among others (e.g., [26], [27], [28], [29], [30], [31]).

These tools implement various techniques for trading precision for performance, usually by reducing variable declarations or instruction precision from double-precision floating-point to single-precision or half-precision floating-point. Some approaches provide rigorous error bounds on the conversion for small kernels while other approaches focus on support for larger-scale applications with fewer formal guarantees. These efforts are all complementary to ours in that they provide additional tools that could be compared or studied using our framework.

There is also rising interest and a growing amount of work in approximate computing in general (of which mixed-precision is a particular approach). [32] and [33] both use approximate floating-point multipliers to drop mantissa bits and improve performance. [34] and [35] present a technique called "perforation" that identifies parts of a computation (e.g., loop iterations) that can be discarded completely to improve performance while providing rigorous accuracy bounds.

Other work has focused on particular domains. A variety of approximate computing techniques in the context of graphics engines and other highly data-parallel domains are presented in [36] and [37]. A heterogeneous architecture using neural-network-based approximations is presented in [38]. Some work has focused on general static analysis techniques. A static analysis and compiler technique for reducing control and memory divergences in fault-tolerant GPU code is proposed in [39]. More recent work [40] contributes a more general "static significance analysis" to guide developers in identifying where an approximation is appropriate. There have also been proposals regarding hardware support for approximate computing. A "scalable effort" hardware is presented in [41] to exploit approximate computing at various levels of abstraction, and [42] proposes a specific ISA extension providing support for approximate arithmetic. Like much of the mixed-precision related work, these efforts are all complementary to ours in that they provide additional approaches to approximate computing that could be compared or studied using our framework.

VII. Conclusion

While approximate computing is emerging as an alternate beyond-Moore computing technique to speedup computational throughput, there is an increasing need to have a benchmark suite to evaluate different approximate computing methods. Mixed-precision methods are one of the most used in the approximate computing domain with several tools and techniques being proposed in the literature. Yet there is no single benchmark suite for the comparison of mixed-precision methods. Most mixed-precision tools are often evaluated on very different workloads or benchmarks and this discrepancy in the evaluation benchmarks that are used in all previous studies precludes the community from having a complete picture of the real effectiveness of mixed-precision tools.

We propose HPC-MixPBench, a benchmark suite of programs for approximate computing analysis. HPC-MixPBench is composed of codes that represent common HPC workloads, some of which are used in the procurement of HPC systems. To demonstrate the capability of the benchmark suite, we evaluate it on several search-based strategies that have been proposed in the literature and report several insights previously unreported about these strategies. Some of the findings of our study are: (a) the behavior of genetic algorithms for mixed-precision search are the easiest to predict, (b) the delta debugging method tends to provide configurations with higher speedups, and (c) preprocessing the application source code to group variables into clusters that share the same precision increases the effectiveness of search algorithms.

We expect that the findings we report in this paper can help HPC application developers choose the appropriate mixed-precision method for their application or workload; reporting such insights was not possible before because of the absence of such a benchmark suite.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments and helpful suggestions. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and was supported by the LLNL-LDRD Program under Project No. 20-ERD-043 (LLNL-CONF-809027).

References

[1] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate computing: A survey," *IEEE Design & Test*, vol. 33, no. 1, pp. 8–22, 2015.

[2] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys (CSUR)*, vol. 48, no. 4, pp. 1–33, 2016.

[3] P. V. Kotipalli, R. Singh, P. Wood, I. Laguna, and S. Bagchi, "Ampt-ga: automatic mixed precision floating point tuning for gpu applications," in *Proceedings of the ACM International Conference on Supercomputing* , 2019, pp. 160–170.

[4] H. Menon, M. O. Lam, D. Osei-kuffuor, M. Schordan, S. Lloyd, K. Mohror, and J. Hittinger, "ADAPT : Algorithmic Differentiation Applied to Floating-Point Precision Tuning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'18)*. Dallas, Texas: IEEE Press, 2018, pp. 48:1–48:13.

[5] I. Laguna, P. C. Wood, R. Singh, and S. Bagchi, "GPUMixer: Performance-Driven Floating-Point Tuning for GPU Scientific Applications," in *High Performance Computing*, M. Weiland, G. Juckeland, C. Trinitis, and P. Sadayappan, Eds. Cham: Springer International Publishing, 2019, pp. 227–246.

[6] 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 Supercomputing (ICS '13)*. New York, New York, USA: ACM Press, Jun. 2013, p. 369.

[7] 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.

[8] M. O. Lam, T. Vanderbruggen, H. Menon, and M. Schordan, "Tool Integration for Source-Level Mixed Precision," in *Proceedings of the Third International Workshop on Software Correctness for HPC Applications Held in Conjunction with SC19: The International Conference for High Performance Computing, Networking, Storage and Analysis* , Denver, CO, 2019.

[9] M. O. Lam and J. K. Hollingsworth, "Fine-Grained Floating-Point Precision Analysis," *International Journal of High Performance Computing Applications*, p. 1094342016652462, Jun. 2016.

[10] 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 (ISSTA'18)*, no. 333. Amsterdam, Netherlands: ACM, 2018, pp. 333–343.

[11] P. V. Kotipalli, R. Singh, P. Wood, I. Laguna, and S. Bagchi, "AMPT-GA: Automatic Mixed Precision Floating Point Tuning for GPU Applications," in *Proceedings of the ACM International Conference on Supercomputing*. New York, NY, USA: ACM, 2019, pp. 160–170, series Title: ICS '19.

[12] S. Anand, A. Orso, and M. J. Harrold, "Type-dependence analysis and program transformation for symbolic execution," in *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 117–133.

[13] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008, pp. 72–81.

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

[15] M. A. Heroux, "Hpccg solver package, version 00," 3 2007. [Online]. Available: https://www.osti.gov//servlets/purl/1230960

[16] C. Rubio-González, D. Hough, C. Nguyen, B. Mehne, K. Sen, J. Demmel, W. Kahan, C. Iancu, W. Lavrijsen, and D. H. Bailey, "Floating-point precision tuning using blame analysis," in *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. New York, New York, USA: ACM Press, 2016, pp. 1074–1085.

[17] 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.

[18] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard, "Quality of service profiling," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 25–34.

[19] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin, "ACCEPT: A Programmer-Guided Compiler Framework for Practical Approximate Computing," Tech. Rep., 2015.

[20] N. Damouche, M. Martel, P. Panchekha, C. Qiu, A. Sanchez-Stern, and Z. Tatlock, "Toward a Standard Benchmark Format and Suite for Floating-Point Analysis," in *Numerical Software Verification: 9th International Workshop, NSV 2016, Toronto, ON, Canada, July 17-18, 2016, Revised Selected Papers*, S. Bogomolov, M. Martel, and P. Prabhakar, Eds. Springer International Publishing, 2017, pp. 63–77.

[21] D. Thien, B. Zorn, P. Panchekha, and Z. Tatlock, "Toward Multi-Precision, Multi-Format Numerics," in *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness)*, Nov. 2019, pp. 19–26.

[22] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran, "Axbench: A multiplatform benchmark suite for approximate computing," *IEEE Design & Test*, vol. 34, no. 2, pp. 60–68, 2016.

[23] E. Darulova and V. Kuncak, "Sound compilation of reals," *ACM SIGPLAN Notices*, vol. 49, no. 1, pp. 235–235–248–248, Jan. 2014.

[24] 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 (ICCPS'18)*. Porto, Portugal: IEEE Press, 2018, pp. 208–219.

[25] E. Darulova, A. Izycheva, F. Nasir, F. Ritter, H. Becker, and R. Bastian, "Daisy - Framework for Analysis and Optimization of Numerical Programs (Tool Paper)," in *24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, vol. 10805 LNCS, Thessaloniki, Greece, 2018, pp. 270–287.

[26] A. A. Gaffar, O. Mencer, W. Luk, P. Y. K. Cheung, and N. Shirazi, "Floating-point bitwidth analysis via automatic differentiation," in *Proceedings - 2002 IEEE International Conference on FieId-Programmable Technology, FPT 2002*, 2002.

[27] R. Nathan, B. Anthonio, S. L. Lu, H. Naeimi, D. J. Sorin, and X. Sun, "Recycled Error Bits: Energy-Efficient Architectural Support for Floating Point Accuracy," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*. IEEE Press, 2014, pp. 117–127.

[28] S. Graillat, F. Jézéquel, R. Picot, F. Févotte, and B. Lathuilière, "PROMISE: Floating-point precision tuning with stochastic arithmetic," in *17th International Symposium on Scientific Computing, Computer Arithmetic and Verified Numerics (SCAN 2016)*, UPPSALA, Sweden, Sep. 2016, pp. 98–99.

[29] N.-M. Ho, E. Manogaran, W.-F. Wong, and A. Anoosheh, "Efficient floating point precision tuning for approximate computing," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2017, pp. 63–68.

[30] 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.

[31] Y. Chatelain, E. Petit, P. de Oliveira Castro, G. Lartigue, and D. Defour, "Automatic Exploration of Reduced Floating-Point Representations in Iterative Methods," in *Euro-Par 2019: Parallel Processing*, ser. Lecture Notes in Computer Science, R. Yahyapour, Ed. Cham: Springer International Publishing, 2019, pp. 481–494.

[32] S. Froehlich, D. Große, and R. Drechsler, "Towards Reversed Approximate Hardware Design," in *2018 21st Euromicro Conference on Digital System Design (DSD)*, Aug. 2018, pp. 665–671.

[33] S. Rehman, W. El-Harouni, M. Shafique, A. Kumar, J. Henkel, and J. Henkel, "Architectural-space exploration of approximate multipliers," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 2016, pp. 1–8.

[34] H. Hoffmann, S. Misailovic, S. Sidiroglou, M. Rinard, and A. Agarwal, "Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures," Tech. Rep. MIT-CSAIL-TR-2009-042, 2009.

[35] M. Rinard, "Probabilistic accuracy bounds for fault-tolerant computations that discard tasks," in *Proceedings of the 20th Annual International Conference on Supercomputing*, ser. ICS '06. Cairns, Queensland,

Australia: Association for Computing Machinery, Jun. 2006, pp. 324–334.

[36] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, “SAGE: Self-tuning approximation for graphics engines,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. Davis, California: Association for Computing Machinery, Dec. 2013, pp. 13–24.

[37] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, “Paraprox: Pattern-based approximation for data parallel applications,” in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’14. Salt Lake City, Utah, USA: Association for Computing Machinery, Feb. 2014, pp. 35–50.

[38] B. Grigorian, N. Farahpour, and G. Reinman, “BRAINIAC: Bringing reliable accuracy into neurally-implemented approximate computing,” in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2015, pp. 615–626.

[39] J. Sartori and R. Kumar, “Branch and Data Herding: Reducing Control and Memory Divergence for Error-Tolerant GPU Applications,” *IEEE Transactions on Multimedia*, vol. 15, no. 2, pp. 279–290, Feb. 2013, conference Name: IEEE Transactions on Multimedia.

[40] S. A. Metwalli and Y. Hara-Azumi, “SSA-AC: Static Significance Analysis for Approximate Computing,” Apr. 2019.

[41] V. K. Chippa, D. Mohapatra, K. Roy, S. T. Chakradhar, and A. Raghunathan, “Scalable Effort Hardware Design,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 9, pp. 2004–2016, Sep. 2014, conference Name: IEEE Transactions on Very Large Scale Integration (VLSI) Systems.

[42] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, “Architecture support for disciplined approximate programming,” in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. London, England, UK: Association for Computing Machinery, Mar. 2012, pp. 301–312.