Precision Tuning in Parallel Applications

Gabriele Magnani
DEIB – Politecnico di Milano, Italy

Lev Denisov
DEIB – Politecnico di Milano, Italy

Daniele Cattaneo
DEIB – Politecnico di Milano, Italy

Giovanni Agosta
DEIB – Politecnico di Milano, Italy

Abstract

Nowadays, parallel applications are used every day in high performance computing, scientific computing and also in everyday tasks due to the pervasiveness of multi-core architectures. However, several implementation challenges have so far stifled the integration of parallel applications and automatic precision tuning. First of all, tuning a parallel application introduces difficulties in the detection of the region of code that must be affected by the optimization. Moreover, additional challenges arise in handling shared variables and accumulators. In this work we address such challenges by introducing OpenMP parallel programming support to the TAFFO precision tuning framework. With our approach we achieve speedups up to 750% with respect to the same parallel application without precision tuning.

2012 ACM Subject Classification Software and its engineering → Compilers; Theory of computation → Parallel computing models

Keywords and phrases Compilers, Parallel Programming, Precision Tuning

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2022.5

Supplementary Material *Software (Source Code)*: `https://github.com/TAFFO-org/TAFFO`

Funding The authors gratefully acknowledge funding from European Union's Horizon 2020 Research and Innovation programme under the Marie Skłodowska Curie grant agreement No. 956090 (APROPOS: Approximate Computing for Power and Energy Optimisation, `http://www.apropos.eu/`).

1 Introduction

Approximate computing is a key addition to the array of techniques that can help in improving the performance and energy efficiency of applications. As such, it has been the subject of significant investigation by the scientific community, in particular in the fields of computer architectures and compilers [24], resulting in a range of different techniques at the software development, compiler, architectural, and circuit level. Within this range, *precision tuning* is particularly interesting for its wide applicability and promising results [7]. This technique aims at exploiting the trade-off between operation accuracy, performance, and energy efficiency that is achieved by manipulating the data types used in each arithmetic operation of a kernel. Typically, in error-tolerant applications where the range of input values is known at compile time, the entire range of values covered by wide floating point representations such as the 64 and 32 bit IEEE754 is unnecessary. This is exploited explicitly by programmers using, e.g., Google's bfloat16, but can more effectively be exploited through an appropriate compiler. Once more, a good amount of research has been performed in

13th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and 11th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2022).
Editors: Francesca Palumbo, João Bispo, and Stefano Cherubin; Article No. 5; pp. 5:1–5:9
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

recent years on the topic of automated management of precision tuning, as shown by a recent survey [5], leading to compiler-based tools such as the *Precimonious* [23], *Daisy* [10], and *TAFFO* [8].

However, in the context of parallel programming models, additional challenges arise that are often not addressed by the abovementioned tools. This is particularly true for tools that perform automatic detection of the region of code to be affected by the precision tuning transformation. Indeed, to guarantee the correctness of the transformed program, precision tuning tools need to reliably detect each parallel region and the sets of variables shared between parallel execution threads. The analysis of the parallel behavior of the program is particularly difficult for languages such as C, C++ and LLVM-IR, which do not support parallel programming paradigms without an auxiliary support library. At the same time, the code transformation steps that are required to implement mixed precision in the program must preserve the correctness of atomic instructions and locking constructs when they are affected by the optimization process.

To address these challenges, in this work we integrate the TAFFO precision tuning plugins for the LLVM compiler framework with the OpenMP support for the same compiler. The rationale for this pairing is explained by the need to produce a proof of concept for precision tuning of parallel programming models. In particular, we focus on TAFFO because it does not work as a source to source, but it is already integrated with the LLVM compiler framework (contrary e.g. to Daisy), and is up to date with recent LLVM versions supporting OpenMP (contrary to Precimonious, which requires a severely outdated version of LLVM). On the side of programming models, OpenMP is one of the simplest models, widely supported by both compilers and benchmarking suites, making it the perfect choice for such a proof of concept.

The rest of this paper is organized as follows. In Section 2 we briefly survey the main tools available for precision tuning during the code compilation stage and the state of parallel language support in such tools. In Section 3 we describe the technical modifications to TAFFO required for supporting OpenMP applications. In Section 4 we provide an experimental evaluation of the system, while in Section 5 we draw some conclusions and highlight future research directions.

2 Related Work

It has been shown that many scientific applications can benefit in terms of performance and energy efficiency from reduced precision calculations [2, 24]. However, the problem of finding the precision mix that satisfies the accuracy requirements while providing the maximum performance is not trivial. As such, automated end-to-end solutions that can perform this process are necessary. The biggest innovation in computer architecture to push performance scaling at the end of Moore's law is parallelization. In recent times, in literature has been studied the question of converting the existing sequential scientific programs into parallel ones [1]. In such cases automated parallelization libraries such as OpenMP [9], MPI [20], and OpenACC are often used. Applying precision mix optimization on top of this parallelization can be beneficial. However, not many of the modern precision tuning tools can work with programs using automated parallelization. In this section, we explore the possibility of automated precision tuning of programs that use OpenMP.

As OpenMP is an automatic parallelization library that supports C/C++ and targets a wide variety of execution platforms, the automated precision tuning tools that can work with C/C++ source code and that do not make assumptions about the target platform are of especial interest for comparison.

Precimonious [23] is a precision tuning tool that works with C/C++ source code and outputs the suggested type changes in a json file. It uses delta-debugging [25] search algorithm to find a precision mix that has better performance while maintaining enough accuracy. Precimonious uses dynamic analysis to verify that the precision mix satisfies the requirements, which depends on having a representative dataset. Precimonious only supports IEEE-754 floating-point types, which limits its use.

CRAFT [16, 15] is a source-to-source precision tuning tool that works with C/C++ code. It uses binary search to determine the precision required at the given program level. It goes through the modules, functions, basic blocks, and individual instructions in a breadth-first search fashion to refine the precision mix. CRAFT uses dynamic analysis to verify that the precision mix satisfies the requirements, which depends on having a representative dataset. The tool can potentially work with OpenMP. CRAFT only supports IEEE-754 floating-point types, which limits its use.

FloatSmith [17, 21] is a source-to-source precision tuning tool that is based on CRAFT [16] and that works with C/C++ code. FloatSmith integrates ADAPT [19] to narrow the search space for CRAFT using static analysis. It uses CRAFT to further optimize the precision mix using different search strategies: combinational, compositional, delta-debugging, hierarchical, hierarchical-compositional, and Genetic Search Algorithm. FloatSmith uses dynamic analysis to verify that the precision mix satisfies the requirements, which depends on having a representative dataset. The paper reports a successful test with OpenMP version of LULESH benchmark [14]. FloatSmith does not support fixed-point types, which limits its use.

GeCoS + ID.Fix [6] is a source-to-source precision tuning tool that works with C/C++ code and targets generic hardware platforms. It uses static analysis technique called value range propagation to infer the value range of dependent variables based on user-annotated variables. However, it mostly focuses on floating point to fixed point conversion to minimise the number of bits used during computation. Additionally, it does not consider the possibility of a mixed precision output, with floating point and fixed point data types coexisting in the same program.

Daisy [10] is a precision tuning tool that targets generic platforms, supports fixed-point types, and provides formal guarantees on the result precision. It uses a combination of mixed-precision tuning with delta-debugging algorithm and rewriting with a genetic algorithm to reduce the roundoff error. Daisy uses a static error analysis with interval arithmetic and SMT [11], and a static heuristic performance cost function. Unfortunately, Daisy requires the program to be written in a Scala-based domain-specific language, and only supports optimization of arithmetic kernels without conditionals or loops, which makes it unsuitable for optimizing programs that use OpenMP.

TAFFO [8] is a precision tuning tool based on LLVM [18] for optimizing C/C++ programs. This paper introduces in TAFFO support for inter-procedural precision tuning of the programs parallelized with OpenMP [9]. TAFFO is a precision tuning tool with user-defined scope based on variable annotations. It performs static code analysis using user-provided range values to infer the algorithm properties and the affected variables and statically validates the effect of the precision tuning step on the target values. It also provides formal guarantees about error magnitude for programs without unbounded loops and gives an estimate when unbounded loops are present. It controls the overhead introduced by the type casting operators [3]. TAFFO is built as an LLVM pass and uses LLVM-IR as its input and output, so it can support a wide variety of programming languages, although it is mainly targeted at programs written in C/C++. It supports optimization using IEEE-754 [13] floating-point, and dynamic fixed-point types with a focus on general-purpose computing platforms.

For the more detailed overview of the field we refer the reader to the recent surveys. Cherubin and Agosta [5] surveys the software tools used at the different stages of precision tuning. Stanley-Marbell et al. [24] introduces unified terminology for quality versus resource usage tradeoffs. It also surveys the field categorizing both software and hardware approaches used on the different levels of the computing stack.

3 Proposed Solution

In this section we describe the modifications we performed on TAFFO to allow handling of OpenMP applications. In order to describe such modifications, we must first give a quick outline of the internal architecture of TAFFO and of the OpenMP support in LLVM.

TAFFO consists of five independent passes, which take the form of a loadable plugin for LLVM-based compilers. The pass-based architecture allows TAFFO to be expandable, easy to use and robust.

The TAFFO tool requires the programmer to define some contextual information related to the value ranges of the inputs and the extent of the area of code that needs to be tuned. This information is inserted through annotation of the source code. The first pass of TAFFO, called *Initializer*, reads such annotations and converts them in the internal data structures required by the rest of TAFFO.

From the user-provided information, TAFFO then analyses the program to conservatively derive the numerical intervals each variable in the program will have at runtime. This pass is called the *Value Range Analysis* or VRA. The information derived by the VRA is then used to determine which reduced-precision data type to use for the variables, a procedure called *Data Type Allocation* (DTA). The DTA can operate based on two different algorithms: a peephole-based algorithm which always chooses the fixed-point data type with the highest valid point position for each variable, and a new optimiser based on ILP techniques [4]. This step is able to optimally mix floating point and fixed point data types by exploiting a mathematical model of how changes to the precision mix affect the speedup and the output error.

Finally, the *Conversion* pass is responsible for applying the data type changes on the program being tuned. The *Feedback Estimator* pass statically analyses the error using state-of-the-art estimation methods [7].

While TAFFO operates at the intermediate representation level, therefore in the so-called *middle-end*, OpenMP is mainly implemented in the compiler frontend. In fact, OpenMP is used by adding specific *pragma* annotations in a C or C++ program. Depending on the pragma, OpenMP will automatically transform what would normally be a non-parallel C language construct into a parallel one. The most common pragmas are the *parallel* pragma and the *for* pragma, and as a result we will focus on supporting such pragmas in our implementation strategy. The *parallel* pragma executes a given code block multiple times in parallel in multiple threads. The number of threads depends on the estimated maximum number of independent threads that can be run on the machine. On the other hand, the *for* pragma must appear before a *for* loop, and it executes each iteration of the loop in parallel with respect to the other. The implementation of a typical OpenMP library uses a fixed thread pool, a well-known implementation strategy for supporting parallel computations while minimizing the operating-system-level overhead of creating and destroying threads every time a new task must be instantiated. The *parallel* pragma starts a new task on each available thread in the pool, all tasks running the same piece of code. The *for* pragma is similar, except that each task executes its body multiple times depending on how many threads are in the pool. Since the trip-count of the loop must be known up-front, the induction variable of the loop shall not be modified in the body of the loop itself.

This functionality is supported by the OpenMP runtime library provided with the CLANG compiler. The creation of the thread pools and the enqueueing of the tasks is performed by code in such library, but the code that calls the library functions is generated by the CLANG frontend at compile-time of the program. The block of code that is associated to each parallel task to be executed is outlined by the CLANG compiler to a separate function. A pointer to this function is then passed to a specific runtime function alongside with the local variables that are used within each thread context. For example, see the C language program in Listing 1 and its corresponding LLVM-IR compiled version in Listing 2. The parallel for statement is translated to a call to a runtime function called "`__kmpc_fork_call`", and the body of the loop is outlined to the function "`.omp_outlined.`". Each thread executes the outlined function, and in that function other runtime calls are present to compute the number of times the body of the loop must be executed.

Listing 1 Example C language OpenMP program.

```
int main()
{
  float container[16];
  #pragma omp parallel for shared(container)
  for (int i = 0; i < 16; i++) {
    float result = i * 0.05;
    container[i] = result;
  }
  return 0;
}
```

Listing 2 Simplified LLVM-IR listing corresponding to the example OpenMP program in Listing 1.

```
define i32 @main() {
entry:
  call void @__kmpc_fork_call(%struct.ident_t* nonnull @2, i32 1,
                              @.omp_outlined., [16 x float]* nonnull %container)
  ret i32 0
}

define internal void @.omp_outlined.(i32* %.global_tid., i32* %.bound_tid.,
                                     [16 x float]* %container) {
entry:
  ...
  call void @__kmpc_for_static_init_4( ... )
  ...
omp.loop.exit:
  call void @__kmpc_for_static_fini( ... )
  ...
  ret void
}
```

In order to add support for OpenMP-aware optimizations in LLVM-IR, an optimization pass must have the appropriate domain knowledge to be able to interpret the meaning of each runtime invocation. Therefore, our implementation approach involved appropriate modifications to the TAFFO passes to add this knowledge. In particular, TAFFO must be able to detect OpenMP outlined functions, and it must be able to infer the trip count of loops in such outlined functions correctly.

To implement these abilities, we modified two passes of TAFFO: Initializer and Conversion. In Initializer, the program is searched for instances of call sites of the OpenMP fork function. At each call site, the OpenMP fork function is temporarily deleted and replaced by a local *trampoline* function, whose body simply calls the OpenMP outlined function. This allows TAFFO's existing code to handle OpenMP programs without additional modifications.

Indeed, analyses and transformations in TAFFO are intra-procedural, and can handle mixed-precision across functions and in function arguments. The VRA and DTA passes are able to inspect each call-site independently and derive mixed-precision data types and value ranges for each call argument. This means that call sites of the same function can have different type annotations depending on the surrounding context. Therefore, the Conversion pass must duplicate each function affected by the mixed-precision transformation a number of times that depends on the number of call sites with unique type assignments. In the final program, as a result, call sites that in the original program invoked the same function now may invoke different functions, depending on whether the call sites now use different types than before or not. This applies to the OpenMP trampoline functions and outlined functions as well. To support the OpenMP runtime, an additional process has been implemented in the Conversion pass, which converts the calls to the trampoline function back to the original call to the OpenMP library function. This procedure effectively also replaces OpenMP outlined functions with their mixed-precision cloned equivalent if needed.

For what concerns the handling of trip count of loops, we exploit the fact that the OpenMP library initialization function of *for* loops takes as arguments the lower bound, upper bound and stride of the loop. Therefore it is easy to analyze the outlined function, detecting the initialization calls and computing the total trip count across all loops with the formula:

$$n = \left\lfloor \frac{u - l + 1}{s} \right\rfloor,$$

where n is the trip count, s is the stride, u is the upper bound and l is the lower bound.

4 Experimental Evaluation

To evaluate our work, we used the PolyBench/C version 3.2 benchmark suite [22], in a version modified for OpenMP support [12]. PolyBench is a collection of several small kernels written in C, covering several computational tasks, such as data mining tasks, linear algebra kernels, BLAS routines and more. Polybench allows to tune the amount of memory to employ for every test in order to be able to adapt to multiple targets, even memory-constrained ones such as microcontrollers.

We run all the benchmarks on a non-uniform memory access (NUMA) server with a 24 Six-Core AMD Opteron(tm) Processor 8435 (2,6 GHz) with 128GB RAM. The operating system is Ubuntu 20.04 LTS. On this machine, not all benchmarks gained a speedup from parallelization. As a result, only a subset of benchmarks were selected, specifically those that can be parallelized without algorithmic changes with respect to the original unmodified PolyBench/C suite, and where parallelization does indeed produce a speedup. These benchmarks are *2mm*, *3mm*, *doitgen*, *gemm*, *syr2k*, and *syrk*.

We compiled the benchmarks in three different configurations, or versions. In all cases, the compiler used was CLANG version 12.0, based on LLVM version 12.0. In the first configuration (denoted with the number zero) both the TAFFO mixed precision optimizations and OpenMP support were disabled, producing a non-parallel benchmark. In the second configuration (denoted with the number 1), OpenMP was enabled, but TAFFO was not used. In the third and

Table 1 Execution time, speedup and average relative error (ARE) data of the non-parallel, parallel, and mixed-precision parallel configurations of the selected subset of PolyBench/C.

Benchmark	t_0 [s]	t_1 [s]	t_2 [s]	S_1	S_2	**ARE**
2mm	105.375	6.199	1.819	1599.9 %	240.7 %	8.85×10^{-9}%
3mm	23.760	1.381	0.803	1620.8 %	71.9 %	7.45×10^{-5}%
doitgen	1.028	0.111	0.080	830.1 %	38.1 %	1.47×10^{-3}%
gemm	101.646	7.515	0.850	1252.6 %	783.7 %	8.85×10^{-9}%
syr2k	6.692	2.595	1.027	157.9 %	152.8 %	8.85×10^{-9}%
syrk	2.285	0.974	0.239	134.6 %	308.3 %	8.85×10^{-9}%

final configuration (denoted with the number 2), both OpenMP and TAFFO were employed. The dataset size – a configuration option provided by all PolyBench/C benchmarks – was set to *normal* for every benchmarks and in every configuration. The benchmarks were instrumented in order to measure the execution time and the error between the TAFFO-optimized mixed precision configuration and the non-mixed-precision configuration.

The data from the experiments conducted as described herein are shown in Table 1. In the table, t_0 refers to the execution time of the non-parallel kernels, t_1 the execution time of the parallel kernels, and t_2 the execution time of the mixed-precision parallel kernels. S_1 is the speedup of the parallel kernel with respect to the non-parallel kernel, and it measures the execution time improvement due to OpenMP support alone. This metric is a percentage value computed with the following formula:

$$S_1 = 100\left(\frac{t_0}{t_1} - 1\right).$$

Similarly, S_2 is the speedup of the mixed-precision parallel kernel (configuration 2) with respect to the parallel kernel (configuration 1), and it quantifies the improvements due to TAFFO's mixed precision optimization. S_2 is computed in the same way as S_1, except replacing t_1 and t_0 with t_2 and t_1 respectively.

Finally, we shown the average relative error (ARE) introduced by the mixed precision optimization performed by TAFFO. To define the ARE, let us represent the output of a benchmark as a vector $X = \{x_1, x_2, ...x_n\}$. If X is the vector of outputs of the unmodified benchmark, and if Y is the vector of outputs of the benchmark optimized by employing TAFFO, the ARE is defined as follows:

$$ARE = \frac{100}{n}\sum_{i=1}^{n}\left|\frac{x_i - y_i}{x_i}\right|.$$

The outputs of the non-parallel configurations and the parallel configurations without mixed-precision are identical, thus we only compare the last mixed-precision parallel configuration with the non-parallel configuration.

The results show that, on top of the already considerable speedup derived from the usage of a parallel algorithm, TAFFO is able to improve the speedup considerably, up to an additional 783% for the *gemm* benchmark. For some benchmarks, namely *syr2k* and *syrk* the gains from mixed precision are comparable with the gains obtainable by this specific parallel implementation. This may partly be due to inefficiencies in the OpenMP runtime implementation itself. None of the benchmarks were slowed-down by the TAFFO mixed precision transformation. Finally, the ARE error metric is under 0.01% for all benchmarks, which is in line with previous results obtained with TAFFO without exploiting OpenMP support. In conclusion, we can state that the OpenMP extension to TAFFO is effective at achieving mixed-precision computation in parallel applications automatically without introducing significant errors with respect to a non-mixed-precision computational kernel.

5 Conclusions

In this work we presented a new extension to the TAFFO precision tuning framework that implements support for the OpenMP parallel programming specification. This extension does not involve modifications to the core analysis passes of TAFFO or the frontend, but it is based on domain knowledge of the OpenMP runtime library. This inherently more scalable approach allows TAFFO to remain target and language independent. We verified the functionality and effectiveness of this extension by applying it on a parallel variant of the well-known PolyBench benchmark suite, achieving speedups up to 750% with respect to the same parallel application without precision tuning.

Further developements involve the further extension of our approach to properly support constructs other than *omp for* and *omp parallel*, such as *omp reduce* and *omp task*. Other goals include support for more parallel programming frameworks and GPU-based programming models such as OpenCL or SYCL.

References

1 Marco Aldinucci, Valentina Cesare, Iacopo Colonnelli, Alberto Riccardo Martinelli, Gianluca Mittone, Barbara Cantalupo, Carlo Cavazzoni, and Maurizio Drocco. Practical parallelization of scientific applications with openmp, openacc and mpi. *Journal of Parallel and Distributed Computing*, 157:13–29, November 2021. `doi:10.1016/j.jpdc.2021.05.017`.

2 Marc Baboulin et al. Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications*, 180:2526–2533, December 2009. `doi:10.1016/j.cpc.2008.11.005`.

3 Daniele Cattaneo, Michele Chiari, Stefano Cherubin, and Giovanni Agosta. Feedback-driven performance and precision tuning for automatic fixed point exploitation. In *International Conference on Parallel Computing*, ParCo, September 2019.

4 Daniele Cattaneo, Michele Chiari, Nicola Fossati, Stefano Cherubin, and Giovanni Agosta. Architecture-aware precision tuning with multiple number representation systems. In *Proceedings of the 58th Annual Design Automation Conference (DAC)*, December 2021 (to appear).

5 Stefano Cherubin and Giovanni Agosta. Tools for reduced precision computation: a survey. *ACM Computing Surveys*, 53(2), April 2020. `doi:10.1145/3381039`.

6 Stefano Cherubin, Giovanni Agosta, Imane Lasri, Erven Rohou, and Olivier Sentieys. Implications of Reduced-Precision Computations in HPC: Performance, Energy and Error. In *International Conference on Parallel Computing (ParCo)*, September 2017.

7 Stefano Cherubin, Daniele Cattaneo, Michele Chiari, and Giovanni Agosta. Dynamic precision autotuning with TAFFO. *ACM Trans. Archit. Code Optim.*, 17(2), May 2020. `doi:10.1145/3388785`.

8 Stefano Cherubin, Daniele Cattaneo, Michele Chiari, Antonio Di Bello, and Giovanni Agosta. TAFFO: Tuning assistant for floating to fixed point optimization. *IEEE Embedded Systems Letters*, 2019. `doi:10.1109/LES.2019.2913774`.

9 Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

10 Eva Darulova et al. Sound mixed-precision optimization with rewriting. In *Proceedings of the 9th ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '18, pages 208–219, 2018. `doi:10.1109/ICCPS.2018.00028`.

11 Eva Darulova and Viktor Kuncak. Sound compilation of reals. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, volume 49(1), pages 235–248, New York, NY, USA, January 2014. Association for Computing Machinery. `doi:10.1145/2578855.2535874`.

12 Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to GPU codes. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10, May 2012.

13 IEEE Computer Society Standards Committee. Floating-Point Working group of the Microprocessor Standards Subcommittee. Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, August 2008. `doi:10.1109/IEEESTD.2008.4610935`.

14 Ian Karlin, Jeff Keasler, and J Robert Neely. Lulesh 2.0 updates and changes. *OSTI – Office of Scientific and Technical Information, U.S. Department of Energy*, July 2013. `doi:10.2172/1090032`.

15 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, 2016. `doi:10.1177/1094342016652462`.

16 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, pages 369–378, New York, NY, USA, 2013. Association for Computing Machinery. `doi:10.1145/2464996.2465018`.

17 Michael O. Lam, Tristan Vanderbruggen, Harshitha Menon, and Markus Schordan. Tool integration for source-level mixed precision. *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness)*, pages 27–35, 2019.

18 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. Int'l Symp. on Code Generation and Optimization*, 2004.

19 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. `doi:10.1109/SC.2018.00051`.

20 Message Passing Interface Forum (MPIF). Mpi: A message-passing interface standard. Technical report, University of Tennessee, USA, 1994.

21 Konstantinos Parasyris et al. Hpc-mixpbench: An hpc benchmark suite for mixed-precision analysis. *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 25–36, October 2020. `doi:10.1109/IISWC50251.2020.00012`.

22 Louis-Noël Pouchet and Tomofumi Yuki. Polybench/C 4.2.1, 2016. URL: `https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/`.

23 Cindy Rubio-González et al. Precimonious: Tuning assistant for floating-point precision. In *Proc. Int'l Conf. on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 27:1–27:12, November 2013. `doi:10.1145/2503210.2503296`.

24 Phillip Stanley-Marbell et al. Exploiting errors for efficiency: a survey from circuits to applications. *ACM Computing Surveys (CSUR)*, 53(3):1–39, 2020.

25 Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002. `doi:10.1109/32.988498`.