Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200054

Feedback-Driven Performance and Precision Tuning for Automatic Fixed Point Exploitation

Daniele CATTANEO Michele CHIARI Stefano CHERUBIN Antonio DI BELLO and Giovanni AGOSTA
Dipartimento di Elettronica, Informazione e Bioingegneria – Politecnico di Milano

Abstract. Precision tuning is an emerging class of techniques that leverage the trade-off between accuracy and performance in a wide range of numerical applications. We employ TAFFO, a compiler-based state-of-the-art framework that relies on fixed point representations to perform precision tuning. It converts floating-point computations into a fixed point version with comparable semantics, in order to obtain performance improvements. Usually, the process of fixed point type selection aims at the minimization of the round-off error introduced by the precision reduction. However, this approach introduces a large number of type cast operations, generating an overhead that may overcome the performance improvements of the conversion to fixed point formats. We propose a control loop architecture that exploits the static analyses provided by TAFFO to reduce the number of type cast operations while keeping the error under a given threshold. We evaluate our approach on three benchmarks of the AXBENCH suite, and we show that in all cases we are able to achieve performance improvements while keeping the introduced numerical error below the given tolerance threshold.

Keywords. precision tuning, fixed point, error estimation, performance estimation

1. Introduction

The scale of computer applications has been steadily increasing across all domains, from embedded systems to High Performance Computing (HPC). In the past, the Dennard scaling and Moore's Law have been the enabling factors for this growth, allowing application developers – especially in High Performance Computing – to reduce the effort spent in fine tuning the resource usage of applications [12]. However, their end has ushered in a new stage in application development, where careful allocation of computational resources is more rewarding than in the past.

In particular, in HPC application development it is common practice to oversize the data types with respect to the accuracy of the results needed by the application. In fact, tuning the size of data types is a time-consuming and error-prone task. In the context of resource-constrained embedded systems, it is customarily performed manually. However, in HPC application development such methods are not feasible due to the scale of the applications and their data sets. To relieve the programmer from this task, we introduced in our previous work [6,4] a compiler-based precision tuning assistant toolchain.

Subsequent evolutions lead us to the development of the state-of-the-art precision tuner based on the LLVM framework, TAFFO [7].

TAFFO performs precision tuning mainly by exploiting the fixed point numerical representation. Fixed point representations are an important resource in application development whenever the need to overcome computational resource limitations emerges. Such representations are predominantly employed in embedded applications. Additionally, they are also exploited as a mean to data size tuning for HPC tasks. TAFFO receives programmer hints as input and it later infers value range information on the data flows in a compilation unit. Then, it transforms the code to use the most appropriate fixed point data types at the intermediate representation level. It is robust enough to support automated conversion for complex C++ benchmarks without rewriting the computational kernels into less expressive languages, such as ANSI C. It is also able to operate both on parallel and serial code.

However, adopting fixed point data types requires fine tuning to achieve performance benefits. In fact, optimizing the allocation of data types to minimize precision loss will impact the execution time, because of the increased number of data type casts, i.e. conversions between different fixed point types. Additionally, we have to consider that some architectures are more suited to fixed point computations than others. To address the challenge of controlling the performance benefits of the floating point to fixed point conversion, we propose as the main contribution of this work a control-loop regulation approach to adjust the adverse effects of the precision tuning task. This control loop leverages a performance and accuracy estimation pass, tailored to the TAFFO toolchain. We call this new step *Feedback Estimator*. The Feedback Estimator is based on a combination of machine learning techniques, and traditional static control flow analyses. The control loop uses the data collected by the Feedback Estimator to improve the floating point to fixed point transformation, by making the chosen precision mix more homogeneous, thus minimizing the number of data type casts. After the aforementioned improvements, the compilation process is repeated, realizing a feedback process between the mixed-precision compiler transformation and the performance evaluation component.

We verify the effectiveness of the Feedback Estimator through a meaningful subset of AXBENCH [23], a well-known approximate computing benchmark suite. In all the benchmarks we considered, we were able to significantly reduce the amount of type cast operations, without significantly compromising the accuracy of the computation, which remains within a satisfactory threshold provided by the user. The reduction of the number of type casts results in a direct reduction of the number of instructions in the program, thus improving the performances of the converted code.

The rest of the article is organized as follows: in Section 2 we describe the main existing solutions concerning this problem, in Section 3 we describe the approach we propose in more detail, in Section 4 we show the results of the application of our technique to selected AXBENCH benchmarks, and we give our concluding remarks in Section 5.

2. Related Works

This work places itself in the prolific field of *reduced precision computation* and in particular, *static precision tuning*. Tools in the state-of-the-art are aimed at automatically producing an optimized version of a given numerical program, that sacrifices computation accuracy to obtain performance gains. Such tools either target the entire program [17,15,19,2], or just computational kernels identified by the user [16,22,20,9]. Performance gains are obtained by using smaller data types, by using fixed point in place of floating point computations, or both. In order to apply this transformation without excessively degrading their accuracy, the precision requirements on the numerical computations must be evaluated, either dynamically [8,19], or statically [9]. An instruction-wise estimation of such requirements allows a very tailored choice of the data types to be used, allowing to minimize data width while keeping a sufficient accuracy. However, this may result in a very heterogeneous precision mix, which requires a very frequent introduction of cast instructions (i.e. bit shifts, when using fixed point types), every time a type mismatch arises in the data flow graph. As a result, the performances of the optimized code degrade, possibly nullifying the gains caused by smaller type width. Also, a high variety of data types in the precision mix may decrease the vectorization opportunities for architectures supporting SIMD instructions.

Different approaches have been proposed in the literature to measure and to limit this overhead. *FRIDGE* [16], *Precimonious* [22], *CRAFT* [18], and *PetaBricks* [2] perform a dynamic estimation of the obtained performance gains by executing and profiling the optimized code on a representative input dataset. This solution may, however, not always be feasible, due to the time required to perform the profiling, or to the unavailability of a sufficiently representative input dataset. Alternatively, the overhead can be estimated a priori via heuristics, such as the number of cast instructions introduced by the code conversion. For example, *FPTuner* [8] exposes a user-defined threshold for the amount of type casts that the tool may insert into the code. This approach has the drawback of requiring a certain skill for the user to pick the threshold. *Autoscaler for C* [17] and other works [19] iteratively optimize the fixed point code by reordering instructions to remove the shift operations whenever possible. *Daisy* [9] estimates the profitability of type transformations by means of a cost function based on the number of cast instructions. Finally, *HiFPTuner* [15] minimizes the number of cast operations by building a data-dependency tree, and trying to assign the same data type to all values in the same cut of the tree.

An excessive reduction of precision mix heterogeneity may severely degrade computation accuracy. To pursue a reasonable trade-off between these two goals, precision tuning tools need to estimate the numerical error introduced by the transformation. *FRIDGE*, *Precimonious*, *CRAFT*, *Autoscaler for C*, and *HiFPTuner* decide whether the accuracy degradation is acceptable by performing an explorative run of the reduced precision version on a representative input dataset. The reliability of this approach depends on the extent to which the validation dataset covers the range of possible real inputs. Other tools, such as *Daisy*, perform a conservative static estimation of the error bounds by means of error propagation techniques.

3. Proposed Solution

We propose an extension of the TAFFO framework that implements a control loop regulation to adjust the effects of the precision tuning task. TAFFO is composed by five stages, namely code pre-processing, value range analysis, data type allocation, code conversion, and feedback estimation. As shown in Figure 1, our control loop design uses the feedback estimation stage to understand whether the proposed mixed precision version should be

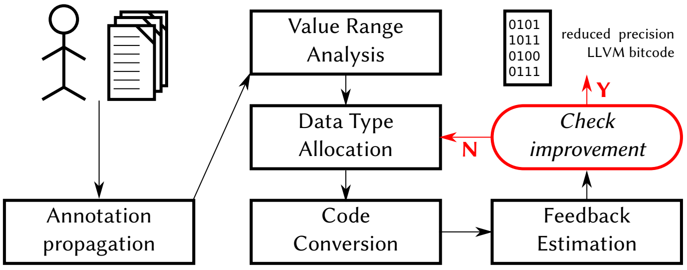

Figure 1. A flow-chart detailing the overall architecture of TAFFO. The extension with the control loop regulation over the latest components of the toolchain is highligthed in red.

improved or not. The control loop acts on the data type allocation stage. In particular, we expose a parameter q from that stage that represents the granularity of the bit partitioning for the fixed point data types.

The goal of this control loop regulation is to maximizing the performance improvement while keeping the error within an acceptable threshold. This task entails the minimization of the number of type cast operations in the final mixed precision code. We consider the number of type casts that are statically present in the program, as opposed to the number of type casts actually executed. The trivial solution of this minimization problem would be an uniform bit partitioning across the whole program. However, the uniform bit partitioning in the data type allocation stage would significantly impact on the error, which may exceed the given threshold. TAFFO already provides a fine-grained bit partitioning in the data type allocation stage. We aim at iteratively reducing the granularity of this allocation to limit the number of bit shift instructions. This new parameter q of the data type allocation can be interpreted as a similarity threshold. Whenever the distance between two fixed point bit partitioning p_1 and p_2 is lower than q, then p_1 and p_2 can be merged into a single bit partitioning p_{12}.

3.1. Similarity distance

Let p_1 and p_2 be two fixed point bit partitionings of the same total width, and let f_1 and f_2 be their respective number of fractional bits, defining the place of the decimal delimiter. The similarity distance between them is defined as $|f_1 - f_2|$. This definition of the distance between two types allows the data type allocator to remove type cast instructions while keeping a limit on the additional error introduced. The order of magnitude of the latter is directly proportional to q, the maximum distance between two types that allows them to be merged into a single one. The resulting type is the one among the two that has the highest number of integer bits (and so the minimum number of fractional bits), so that no potential overflows are introduced.

3.2. The Feedback Analyses

TAFFO implements two kinds of analyses in its feedback estimation step. The first one is a functional analysis of the mixed precision code. It is named *Error estimation* and it evaluates the impact of the round-off error due to the real number representation for each intermediate and output value. It propagates rounding errors represented as Affine Forms [11,10], based on the variable ranges estimated by the value range analysis component. The second analysis classifies the mixed precision version on the basis of the expected speedup with respect to the original version. This *performance estimation* step predicts whether the mixed precision version is going to be much slower (speedup < 0.8), much faster (speedup ≥ 1.2), or almost the same of the original version. It is based on a *Gradient Boosting* classifier [14], provided by the machine learning framework `scikit-learn` [21].

Although the error estimation provides a conservative over-approximation of the round-off error, it captures the trend of the actual round-off error at runtime. Figure 2, Figure 3, and Figure 4 reflect this property. We want to save compilation time by avoiding the code generation and execution of mixed precision versions that are likely to be not profitable. Therefore, the minimization of the estimated error is a good proxy for the minimization of the actual error at runtime. On the contrary, the TAFFO performance estimation only provides a coarse-grained classification. The difference between the optimal solution and a solution that is close to the optimal is not likely to be captured by this classification. Thus, the TAFFO classification does not represent a metric which is sufficient to drive the regulator from the performance point of view.

We compute the number of type cast instruction that are removed by the merge of fixed point bit partitioning in the data type allocation stage by using the exposed parameter q. This metric is monotonous non-decreasing with respect to q. As this metric represents the number of instructions that were removed from the application, we design an heuristic regulation function that assumes a positive correlation between the number of removed type cast and the speedup.

3.3. Regulation Policy

The purpose of the regulation policy is to try to achieve a significant speedup, while maintaining the error within acceptable bounds. The user is required to provide a bound e_{max} for the maximum acceptable absolute error on the output values. Then, two different settings are available for the policy: it can be set to either maximize speedup, or minimize error. In the former case, it explores values of q starting from $q = 32$, and decreases q until the estimated error becomes lower than e_{max}. If the speedup is deemed negative at $q = 32$, or if it is still negative when the error reaches e_{max}, it means it is not possible to achieve a speedup while keeping the error acceptable. In this case, the program is not converted to fixed point format. The pseudocode in Algorithm 1 formalizes this description.

Algorithm 1.: Performance Maximization

```
q ← 32
error ← estimate_error(q)
speedup ← estimate_speedup(q)
while error > $e_{max}$ and speedup == faster do
    q ← q − 1
    error ← estimate_error(q)
    speedup ← estimate_speedup(q)
end while
if speedup == faster and error ≤ $e_{max}$ then
    return q
else
    return −1
end if
```

Algorithm 2.: Error Minimization

```
q ← 0
error ← estimate_error(q)
speedup ← estimate_speedup(q)
while error ≤ $e_{max}$ and speedup ≠ faster do
    q ← q + 1
    error ← estimate_error(q)
    speedup ← estimate_speedup(q)
end while
if speedup == faster and error ≤ $e_{max}$ then
    return q
else
    return −1
end if
```

The setting that minimizes error is essentially symmetric, since it starts from $q = 0$, and it increases it until the estimated speedup becomes greater than 1.2, while the error remains $\leq e_{max}$, as we can see in Algorithm 2.

4. Evaluation

We evaluated our feedback-driven approach on three benchmarks form AXBENCH [23], a popular approximate computing benchmark suite. The benchmarks we chose, which are the implementations of real-world numerical algorithms from different domains, are *Black-Scholes*, *FFT* and *K-means*. Below we describe the results we obtained for each benchmark, and the behavior of the regulation policy.

4.1. Black-Scholes

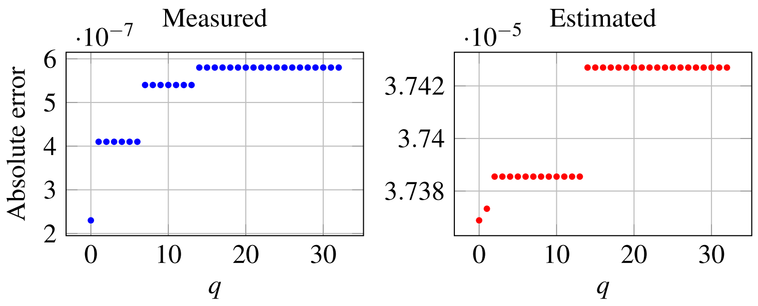

Figure 2. Measured and estimated error for the *Black-Scholes* benchmark.

Black-Scholes is a financial application that numerically computes the equation for the value of European call options according to the Black-Scholes model of a financial market. Its input dataset consists of 48,000 options. The accuracy of the optimized version is evaluated by computing the average absolute error of its output with respect to the floating point version.

Figure 2 shows the measured (left) and estimated (right) absolute errors with respect to parameter q. Note that the feedback analysis overestimates the absolute error by two orders of magnitude, which is in line with other results obtained with the technique we used [10]. Nevertheless, the estimated error consistently follows the shape of the measured one when varying parameter q. In the few cases it does not, the error bound is still conservative. Thus, it is possible to use it to tune parameter q, in order to improve performance. The performance estimator predicts a positive speedup for all values of q. If the regulation policy is set to maximize accuracy, the framework chooses $q = 0$ as the final parameter setting. If, on the contrary, it is set to maximize performance, it chooses $q = 32$, as the estimated error remains acceptable.

The number of removed casts, which is shown in Figure 5, increases with q, and its variation with respect to q is consistent with the absolute error. When $q = 32$, all casts are removed, which ensures that there is a performance improvement, due to the lower number of instructions involved in the computation. In all benchmarks, the maximum value of q is 32, because this is the width of all fixed point data types used.

Figure 6 shows the relation between the number of removed casts and the measured relative error on the output. Clearly, from the point of view of numerical accuracy Black-Scholes is not very sensitive to the removal of cast instructions, as its relative error remains well below 1%, even when removing all casts. This allows the optimized version of the benchmark to achieve the maximum performance improvement.

4.2. FFT

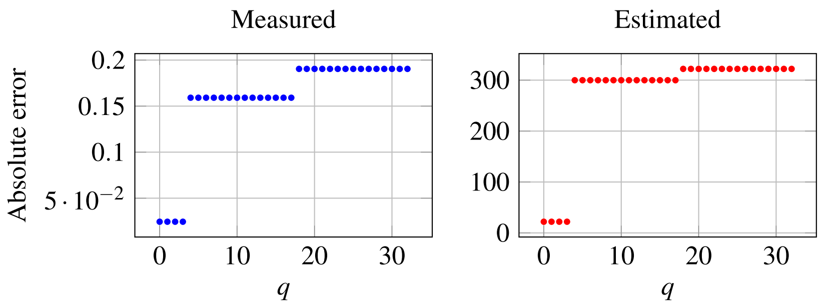

Figure 3. Measured and estimated error for the *FFT* benchmark.

FFT is an implementation of the Radix-2 Cooley-Tukey Fast Fourier Transform, an algorithm widely used in signal processing. It receives as an input signal a discrete rectangular wave of period K and duty cycle 1% in the time domain, and converts it into the frequency domain. Again, the output accuracy is measured by computing the absolute error.

The measured and estimated errors are reported in Figure 3. This time, the estimated error becomes extremely high for $q \geq 4$, exceeding the user-defined error threshold, which is $e_{max} = 50Hz$, corresponding to a relative error around 1%. The regulation policy chooses $q = 3$ when optimizing performance, thus removing around 19% of cast instructions. This is a rather significant improvement, even if the value of q remains low. Figure 6 shows how the measured relative error approaches and becomes greater than 1% as the amount of removed casts gets higher than around 19%.

Instead, $q = 0$ is chosen when optimizing error, since the speedup due to the sole conversion of floating point computations to fixed point types is still estimated as high.

Note that, according to Figure 5, even with $q = 32$, only 37.5% of the cast instructions are removed. This is due to the fact that the data type allocation stage always refrains from merging two types when this operation could potentially cause overflows during the execution, according to the value ranges estimated for each variable. This makes sure the accuracy reductions due to the optimization are gradual, and do not compromise the correctness of the program completely.

4.3. K-means

K-means uses a popular machine learning algorithm to classify pixels from an image into a user-specified number of clusters. As an input dataset for its evaluation, we use the one provided by AxBench, i.e. a set of RGB pictures. The error introduced by the fixed-point optimization is measured and estimated on the Euclidean distance between single

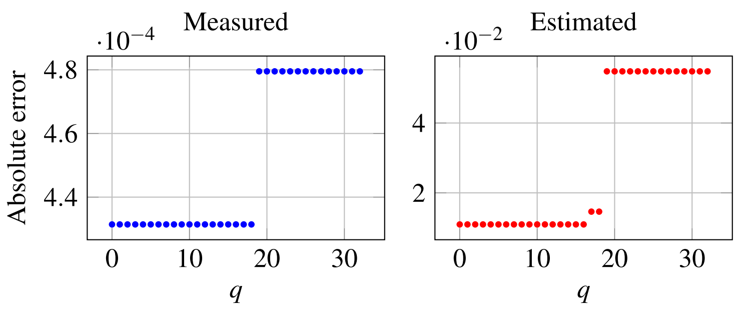

Figure 4. Measured and estimated error for the *K-means* benchmark.

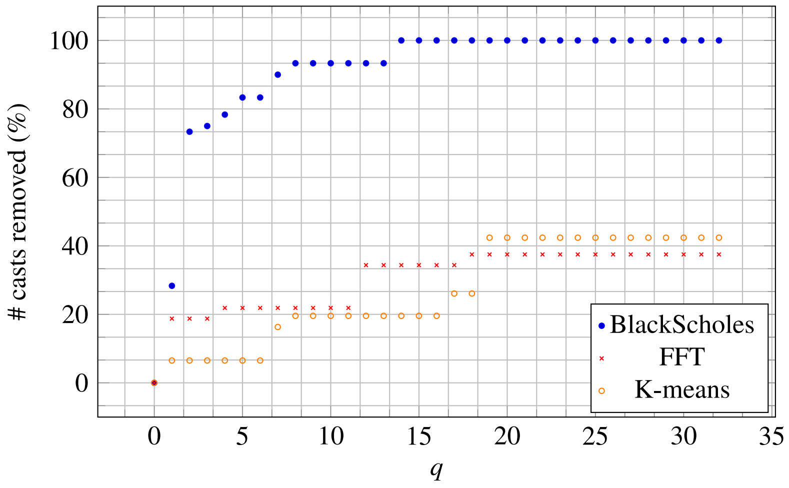

Figure 5. Percentage of type cast instructions removed for all valid values of q, with respect to the number of type casts when $q = 0$.

pixels and cluster centroids. The distance results from the main computational kernel of the application, and it determines the classification of each pixel in its cluster. Its error is thus significantly representative of the cluster misclassification rate introduced by the optimization.

In this case, the only significant change in accuracy occurs between $q = 18$ and $q = 19$, for both the measured and the estimated error. The chosen error threshold is $e_{max} = 2 \cdot 10^{-2}$, and the speedup is always estimated greater than 1.2. Therefore, the policy chooses $q = 0$ when optimizing for accuracy, and $q = 18$ when optimizing for speedup, thus removing 26% of the cast instructions. The number of removed cast instruction is sensible for this benchmark, too. Again, a number of cast instructions cannot be removed even with $q = 32$, due to overflow concerns.

5. Conclusion

In this paper, we presented a major extension of the TAFFO framework for precision tuning. In particular, we introduced a control loop for data type selection, which is governed by a feedback estimation component. The proposed modifications enable TAFFO to re-

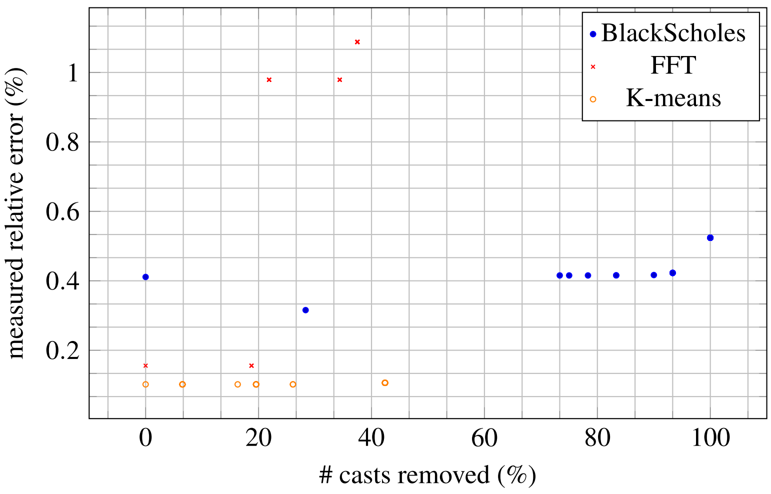

Figure 6. Measured relative error with respect to the amount of cast instructions removed.

duce the number of data type casts. This effect is achieved by merging similar fixed point configurations whenever the impact of such merge is zero or particularly small. This feature enables significant performance improvements. An experimental campaign carried out on the AXBENCH benchmark suite for approximate computing, restricted to the benchmarks that include significant floating point computations, shows the effectiveness of the proposed approach. Indeed, TAFFO is able to explore the approximation options, to correctly estimate the error introduced by different levels of optimization (corresponding to the aggressiveness of the data type casts removal), and to identify the best solution in performance at the requested accuracy level. As a result, when imposing an accuracy threshold of 1% numerical error, TAFFO produces an optimized version with a number of cast instructions between 19% and 100% lower than the baseline version with the default precision mix, resulting in a performance speedup, due to the reduced number of instructions.

As future development, we plan to extend the set of data types managed by TAFFO with the half-precision floating point `bfloat16` [1] and arbitrary precision data types [13,3]. This extension entails the porting of all the analyses and transformations to generic data types, but will expand usefulness of TAFFO to many HPC use cases, such as simulations of chaotic systems. An additional development line involves the coupling of the TAFFO framework with a dynamic partial re-compilation framework such as LIBVC [5] to implement dynamic precision tuning.

References

[1] BFLOAT16 – Hardware Numerics Definitions. Technical report, Intel Corporation, Nov 2018.

[2] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 85–96, April 2011.

[3] D. H. Bailey. A thread-safe arbitrary precision computation package (full documentation). Technical report, Mar 2017.

[4] D. Cattaneo, A. Di Bello, S. Cherubin, F. Terraneo, and G. Agosta. Embedded operating system optimization through floating to fixed point compiler transformation. In *21st Euromicro Conference on Digital System Design (DSD)*, pages 172–176, Aug 2018.

[5] S. Cherubin and G. Agosta. libVersioningCompiler: An easy-to-use library for dynamic generation and invocation of multiple code versions. *SoftwareX*, 7:95 – 100, 2018.

[6] S. Cherubin, G. Agosta, I. Lasri, E. Rohou, and O. Sentieys. Implications of Reduced-Precision Computations in HPC: Performance, Energy and Error. In *International Conference on Parallel Computing (ParCo)*, Sep 2017.

[7] S. Cherubin, D. Cattaneo, M. Chiari, A. Di Bello, and G. Agosta. TAFFO: Tuning assistant for floating to fixed point optimization. *IEEE Embedded Systems Letters*, 2019.

[8] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić. Rigorous floating-point mixed-precision tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 300–315, 2017.

[9] 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, pages 208–219, 2018.

[10] E. Darulova and V. Kuncak. Trustworthy numerical computation in scala. *SIGPLAN Not.*, 46(10):325–344, Oct. 2011.

[11] L. H. de Figueiredo and J. Stolfi. Affine arithmetic: Concepts and applications. *Numerical Algorithms*, 37(1):147–158, Dec 2004.

[12] M. Duranton, K. De Bosschere, B. Coppens, C. Gamrat, M. Gray, H. Munk, E. Ozer, T. Vardanega, and O. Zendra. *The HiPEAC Vision 2019*. HiPEAC CSA, Jan. 2019.

[13] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. Mpfr: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2), June 2007.

[14] J. H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[15] 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 2018, pages 333–343, New York, NY, USA, 2018. ACM.

[16] H. Keding, M. Willems, M. Coors, and H. Meyr. FRIDGE: A fixed-point design and simulation environment. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '98, pages 429–435, 1998.

[17] K.-I. Kum, J. Kang, and W. Sung. AUTOSCALER for C: an optimizing floating-point to integer C program converter for fixed-point digital signal processors. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 47(9):840–848, Sept 2000.

[18] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. Legendre. Automatically adapting programs for mixed-precision floating-point computation. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 369–378, 2013.

[19] D. Menard, D. Chillet, F. Charot, and O. Sentieys. Automatic floating-point to fixed-point conversion for dsp code generation. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '02, pages 270–276, 2002.

[20] R. Nobre, L. Reis, J. a. Bispo, T. Carvalho, J. a. M. P. Cardoso, S. Cherubin, and G. Agosta. Aspect-driven mixed-precision tuning targeting gpus. In *Proceedings of the 9th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and the 7th Workshop on Design Tools and Architectures For Multicore Embedded Computing Platforms*, PARMA-DITAM '18, pages 26–31, Jan 2018.

[21] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[22] 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 on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 27:1–27:12, Nov 2013.

[23] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran. AxBench: A multiplatform benchmark suite for approximate computing. *IEEE Design Test*, 34(2):60–68, April 2017.