Using Approximate Computing for the Calculation of Inverse Matrix p-th Roots

Michael Lass, Thomas D. Kühne and Christian Plessl, *Senior Member, IEEE*

***Abstract*—Approximate computing has shown to provide new ways to improve performance and power consumption of error-resilient applications. While many of these applications can be found in image processing, data classification or machine learning, we demonstrate its suitability to a problem from scientific computing. Utilizing the self-correcting behavior of iterative algorithms, we show that approximate computing can be applied to the calculation of inverse matrix p-th roots which are required in many applications in scientific computing. Results show great opportunities to reduce the computational effort and bandwidth required for the execution of the discussed algorithm, especially when targeting special accelerator hardware.**

***Index Terms*—Approximate computing, Iterative methods, Linear algebra, Scientific computing.**

I. Introduction

APPROXIMATE computing has gained a lot of attention over the last years as a technique to accelerate systems or applications or increase their power efficiency by allowing errors or a loss of precision. Many different techniques have been proposed, such as bit truncation, approximate adder circuits, analog computing and usage of neural networks. Typical applications for approximate computing are from the areas of image processing, data classification and machine learning where small variations in the output are barely noticeable and can be tolerated.

Chippa et al. [1] showed that additionally, iterative algorithms are often inherently resilient to approximation because errors introduced in one iteration are likely to be fixed in later iterations. Iterative algorithms can be found for many numeric problems encountered in scientific computing. Klavik et al. [2] demonstrated that the *Conjugate Gradient* method used to solve systems of linear equations performs well using low precision arithmetic for the computationally expensive parts. Schöll et al. [3], [4] describe the application of efficient fault-tolerance to Preconditioned Conjugate Gradient in order to execute this algorithm on approximate hardware. In this work we focus on an iterative algorithm which is used to calculate the inverse p-th root $A^{-1/p}$ for a given symmetric positive definite matrix A.

Calculating inverse p-th roots is important for a large variety of different applications in scientific computing such as preconditioning, Kalman filtering and solving linear least squares problems, where approximation may generally be sufficient, as well as non-linear optimization, solving systems of linear equations and solving generalized eigenvalue problems, particularly Schrödinger and Maxwell equations, which need to be solved exactly. Nevertheless, as will be demonstrated in the following, under certain circumstances precise results can be obtained despite the presence of approximation.

In this work, we contribute an analysis of the error resiliency of the given algorithm and discuss the potential benefits of using low-precision arithmetic and data representation in applications using this algorithm to calculate inverse p-th roots of large matrices.

Manuscript received January 13, 2017; revised May 30, 2017; accepted September 27, 2017. Date of publication XXXXXXX XX, XXXX; date of current version XXXXXXX XX, XXXX.

M. Lass and C. Plessl are with the Department of Computer Science and the Paderborn Center for Parallel Computing, Paderborn University, 33098 Paderborn, Germany (e-mail: firstname.lastname@uni-paderborn.de).

T.D. Kühne is with the Department of Chemistry and the Paderborn Center for Parallel Computing, Paderborn University, 33098 Paderborn, Germany (e-mail: tdkuehne@mail.uni-paderborn.de).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier XX.XXXX/XXX.XXXX.XXXXXXX

II. Calculation of Inverse Matrix p-th Roots

The iterative algorithm discussed in this work was first described by Bini et al. [5]. We briefly describe the algorithm in the following. Let $C_k,\ k = 0, 1, \ldots$ be the sequence of intermediate result matrices. Starting with an initial guess C_0, in each iteration the result is refined by

$$C_{k+1} = \frac{1}{p}\left((p+1)C_k - C_k^{p+1}A\right) \tag{1}$$

If the initial guess C_0 was already close to $A^{-1/p}$ such that

$$\left\|I - C_0^p A\right\|_2 < 1 \tag{2}$$

C_k for $k \to \infty$ converges against $A^{-1/p}$. For $p = 1$ the algorithm corresponds to the well-known Newton-Schulz method [6] used to iteratively calculate inverse matrices.

There are different possible choices for C_0. For our evaluation we use

$$C_0 = \left(\|A\|_1 \cdot \|A\|_\infty\right)^{-1} A^\mathsf{T} \tag{3}$$

which is proven to always fulfil constraint (2) and therefore guarantees convergence [7].

III. Use-Cases for Approximation

Approximate computing is applicable to the presented algorithm in different use-cases. Some applications benefit from having an approximation of inverse p-th roots if these can be

Published in: *IEEE Embedded Systems Letters*
© IEEE 2017, DOI: 10.1109/LES.2017.2760923

calculated quickly. However, even if exact results are required, the iterative nature of the algorithm allows using imprecise arithmetic for the majority of iterations and then refining the result in few iterations using precise arithmetic. There are different scenarios that suggest using approximate arithmetic or approximate storage which are discussed in the following.

A. Approximate Arithmetic

Approximating the arithmetic operations in the presented algorithm or using low precision can be beneficial for overall performance or the energy efficiency, depending on the underlying compute platform. Recent GPUs targeting the data center, such as the *NVIDIA Tesla P100*, support half-precision floating-point operations, doubling the peak performance compared to single-precision arithmetic and quadrupling it compared to double-precision arithmetic [8].

For custom hardware accelerators, the data width also plays an important role. Currently, FPGAs are becoming more and more common as an accelerator platform, next to CPUs and GPUs. Reduced precision can significantly lower resource requirements on FPGAs, e.g. for current Xilinx devices using less than 17 stored mantissa bits reduces the number of required DSPs by half compared to single-precision. For custom CMOS designs it has been shown that the power consumption of multipliers rises at least quadratically with the number of input bits [9]. Because the delay increases too, this has an amplified effect on the energy consumption. Using fixed-point arithmetic with only few bits can further simplify custom designs for FPGAs or ASICs.

Lastly, more advanced approximate computing techniques such as biased voltage over-scaling, approximate arithmetic circuits or specifically targeted overclocking have the potential to improve clock speeds or lower the complexity and power consumption of custom hardware designs.

B. Approximate Storage

In applications using the examined algorithm, data sets are typically very large, i.e. calculations need to be performed on matrices containing billions of entries. This not only leads to great computational demands but also requires large memories and high memory bandwidth. Even if calculations are performed precisely, the required memory space and bandwidth can be reduced by storing intermediate results after each iteration as well as the input data itself with less precision, using fewer bits.

If the computations are to be performed on special hardware accelerators such as GPUs or FPGAs, the data needs to be transferred between the host and these devices. In this case, bandwidth quickly becomes a limiting factor, motivating the use of low precision representations for the transferred data.

In the scope of this work, we evaluate the impact of both, low precision arithmetic and low precision storage of intermediate results, on the examined algorithm.

IV. Evaluation

To assess the resiliency of the given algorithm to certain errors, we simulate simple approximation techniques. First, we use custom-precision floating-point and fixed-point arithmetic for *all involved calculations* in order to get an understanding of the required data ranges and precision. Second, we restrict the use of these data types to the input data and stored intermediate results, simulating *approximate storage* or *data exchange* of the involved matrices.

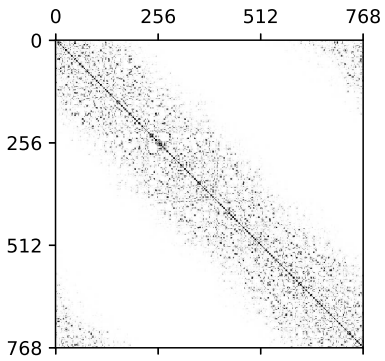

Figure 1. Structure of an examined overlap matrix S ($N = 768$).

A. Problem and Data Set

We use the examined algorithm to efficiently compute two particularly time-consuming kernels of effective single-particle Schrödinger equations, which are to orthogonalize and solve the corresponding generalized eigenvalue problem. For that purpose, we first calculate the inverse square-root $S^{-1/2}$ of the so-called overlap matrix S, which is of dimension $N \times N$ and whose elements read as $S_{ij} = \langle \varphi_i | \varphi_j \rangle$, where φ_i are the N non-orthogonal basis functions spanning the Hilbert space, and subsequently S^{-1} to efficiently solve the resulting orthogonalized matrix eigenvalue equation [10], [11].

Overlap matrices of different sizes have been obtained using a Daubechies Wavelet-based density functional theory code [12], simulating a system of liquid water. Figure 1 shows an overlap matrix S of size $N = 768$ which we used for our evaluation. It is a symmetric positive definite matrix with 25% non-zero elements in the range of $[-1, 1]$, representing a system of 128 H_2O molecules. For larger matrices density decreases to 12.4% ($N = 1536$), 6.2% ($N = 3072$) and 3.1% ($N = 6144$) non-zero elements. This is a manifestation of the *nearsightedness* principle of electronic matter and the foundation of linear scaling electronic algorithms [13].

B. Methodology

For the presented simulations we use Python along with NumPy and SciPy [14], which provide the required data structures and numeric operations. This allows us to define entirely custom data types, e.g. floating-point types with a custom number of bits in the exponent and mantissa, and fixed-point types with a selectable number of bits and selectable scaling factor. Implementing basic arithmetic operations like addition, subtraction and multiplication for our custom data types enables NumPy to use these data types in its own array data structures and numeric operations.

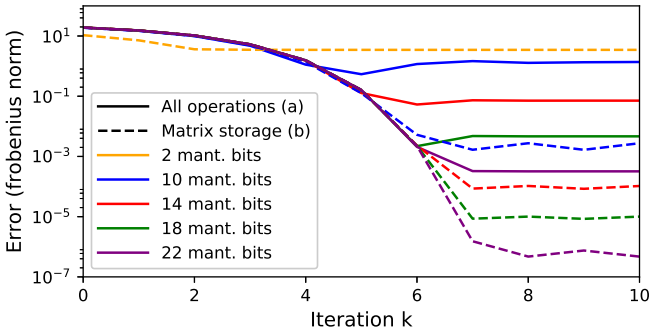

Figure 2. Convergence behavior when using custom-precision floating-point for (a) all arithmetic operations and (b) only for storage of intermediate results.

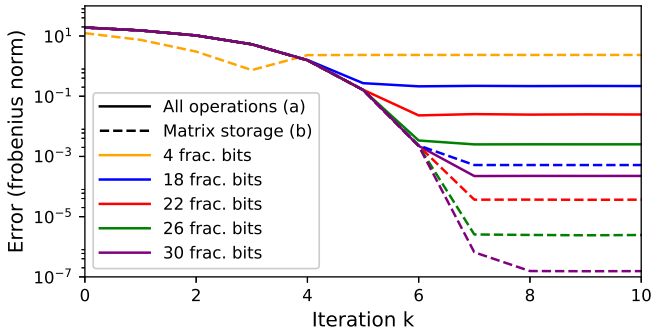

Figure 3. Convergence behavior when using custom-precision fixed-point for (a) all arithmetic operations and (b) only for storage of intermediate results.

This approach allows a very flexible and fast implementation of simulators for different approximation techniques. Besides the mentioned floating-point and fixed-point data types, influences like noise or random bitflips can be easily implemented and adjusted. This flexibility comes at the cost of a performance penalty as each arithmetic operation now implies doing a function call, performing the necessary simulation steps and instantiating a return object. We deal with this performance degradation by implementing all classes in Cython [15], producing statically typed C-code which executes orders of magnitude faster than interpreted Python code, and distributing the simulations over many machines of a large compute cluster.

C. Results

1) Overall Error Resiliency: To assess the overall error resiliency of the algorithm, we initially choose an overlap matrix of dimension $N = 768$ and set $p = 2$ to calculate the inverse square root for these matrices, which is one step of solving the generalized eigenvalue problem as described in Section IV-A. With simulation, we determine the convergence of the algorithm, depending on the given precision. The iterative algorithm shows to be rather resilient to low precision, both for storage of intermediate matrices as well as for all used arithmetic operations.

Figure 2 shows the error between the intermediate solutions obtained from the algorithm using floating-point with custom mantissa widths and a solution that was precomputed using double-precision. As error metric we use the Frobenius norm

$$\left\| C_k - S^{-1/p} \right\|_F := \sqrt{\sum_{i=1}^{N} \sum_{j=1}^{N} |\gamma_{ij} - \chi_{ij}|^2}, \tag{4}$$

where γ_{ij} are the elements of C_k and χ_{ij} those of $S^{-1/p}$.

The observed convergence of the algorithm can be split into two phases: First, the error steadily decreases according to the algorithm's quadratic order of convergence [5]. In the second phase, being limited by the given precision of the data type, the algorithm does not converge further but oscillations may be observed. This shows that the convergence in the first phase is barely influenced by the introduced errors. Only for less than 10 mantissa bits the algorithm does not converge at all. Consequently, half-precision floating-point arithmetic is sufficient to retain convergence. Approximation does however increase the lower bound for the error. Therefore, the second phase of conversion starts earlier for lower precision.

Increasing the precision in later iterations allows the algorithm to further converge against a lower error, opening the possibility for dynamic precision scaling. Observing the changes introduced in each iteration, the necessity of increased precision can be detected at runtime. Note that the use of low precision arithmetic in the first iterations does not increase the overall number of iterations, even when using higher precision in later iterations to achieve more precise results. This opens up gains in performance or energy-efficiency as soon as a single iteration can be executed more efficiently using approximation techniques. The gain in energy-efficiency G can be estimated by

$$G = \frac{E_{\text{prec}} \cdot \#\text{iterations}_{\text{orig}}}{E_{\text{prec}} \cdot \#\text{iterations}_{\text{prec}} + E_{\text{approx}} \cdot \#\text{iterations}_{\text{approx}}} \tag{5}$$

where E_{prec} (E_{approx}) denotes the energy consumption of a precisely (approximately) performed arithmetic operation.

Approximating only the storage of intermediate results allows significantly stronger approximation while achieving similar precision in the output. E.g., storing only 10 mantissa bits allows a similar error as doing all calculation using 18 mantissa bits. In our use case the algorithm still converges if only two mantissa bits are used for all stored values.

Figure 3 shows similar behavior when using fixed-point arithmetic with low precision. To retain convergence, 18 fractional bits are required for arithmetic operations. Again, restricting the approximation to stored intermediate results permits stronger approximation. In our evaluation, storing only four fractional bits showed to be sufficient to retain convergence.

2) Influence of the Matrix Size: Most of the results presented before apply directly to larger matrices from our problem set, in particular when only approximating the storage of intermediate results. Approximating all arithmetic operations using low-precision fixed-point arithmetic however exhibits a limitation. As shown in Figure 4, using 18 fractional bits is sufficient to retain convergence for $N = 768$ but for $N = 1536$ the error eventually increases. The reason for this behavior is that larger matrices from our set are more

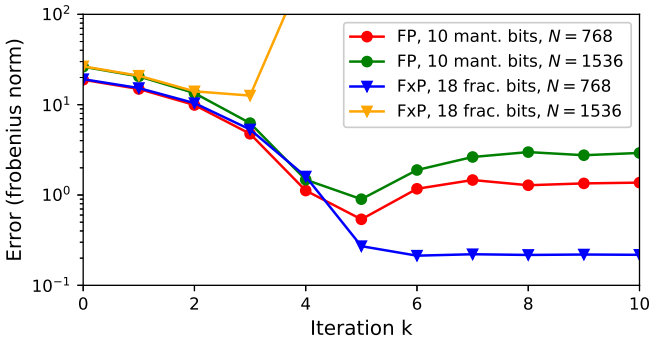

Figure 4. Convergence behavior using custom-precision floating-point (FP) and fixed-point (FxP) for different matrix sizes.

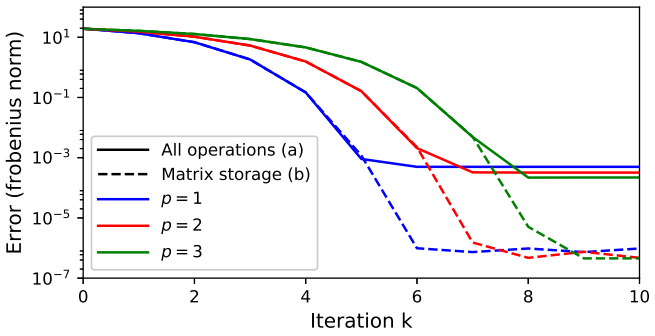

Figure 5. Convergence behavior for different p using custom-precision floating-point with 22 mantissa bits.

sparse (see Section IV-A) and therefore their inverse contain smaller values which cannot be represented appropriately with the given number of fractional bits.

For floating-point this effect is not relevant, as depicted in Figure 4. The slightly larger final error for $N = 1536$ can be explained by the use of the Frobenius norm as error metric since it adds up the quadratic errors of all matrix elements.

3) Influence of p: Calculating the inverse p-th root for $p \neq 2$ shows similar behavior as for $p = 2$, as shown in Figure 5 for custom-precision floating-point arithmetic and storage. With increasing p the algorithm in general needs an increasing number of iterations to converge. This effect is independent of the applied approximation.

4) Influence of the Matrix Condition: The condition of the overlap matrices S depends on the system, in particular the element, that is simulated. The matrices for systems of H_2O molecules used in our evaluation have condition numbers around $\kappa = 1.5$. Calculating the inverse p-th root of matrices with larger condition numbers requires overall more iterations, as discussed by Richters et al. [7]. Additionally, the resulting matrix is expected to be full so that exploiting sparsity of the matrix becomes more difficult. Investigating a relationship between the condition number of the input matrix and the error resiliency of the algorithm remains subject of future work.

V. Conclusion

The presented results show the resiliency of the examined algorithm against errors introduced due to low precision arithmetic and storage. While a certain precision has to be provided to retain convergence of the algorithm, further precision is only required in final iterations if a precise result is desired.

It stands out that the number of iterations required to reach a certain precision does not significantly increase with the amount of approximation. This sets the examined algorithm apart from other iterative methods like the preconditioned conjugate gradient method which was modified to run on approximate hardware by Schöll et al. [4] and showed to require additional iterations when using approximation.

This opens up great opportunities for the acceleration of applications in the scientific computing domain requiring the calculation of inverse p-th roots: Using half-precision floating-point in the first iterations can lead to a $2\times$ speedup for these iterations on suitable GPUs. Resource requirements on FPGAs can be reduced by half and for custom CMOS designs, power consumption of the multipliers can be reduced by a factor of $4\times$. Moreover, the overhead required for data exchange when using GPUs or custom hardware can be significantly reduced as data can be represented using low precision data types.

If a precise solution for $A^{-1/p}$ is required, results obtained using approximation can be refined into a precise solution in very few additional iterations. In a scenario using approximate hardware accelerators, this can be done in software while leaving the main part of the work to the external accelerators.

References

[1] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and Characterization of Inherent Application Resilience for Approximate Computing," in *Des Aut Con*. ACM, 2013.
[2] P. Klavík, A. C. I. Malossi, C. Bekas, and A. Curioni, "Changing Computing Paradigms Towards Power Efficiency," *Philos T Roy Soc A*, vol. 372, no. 2018, 2014.
[3] A. Schöll, C. Braun, M. A. Kochte, and H. J. Wunderlich, "Low-overhead fault-tolerance for the preconditioned conjugate gradient solver," in *IEEE Int Symp Defect*, Oct. 2015, pp. 60–65.
[4] A. Schöll, C. Braun, and H. J. Wunderlich, "Applying efficient fault tolerance to enable the preconditioned conjugate gradient solver on approximate computing hardware," in *IEEE Int Symp Defect*, Sep. 2016, pp. 21–26.
[5] D. A. Bini, N. J. Higham, and B. Meini, "Algorithms for the matrix pth root," *Numer Algorithms*, vol. 39, no. 4, pp. 349–378, 2005.
[6] G. Schulz, "Iterative Berechung der reziproken Matrix," *ZAMM-Z Angew Math Me*, vol. 13, no. 1, pp. 57–59, 1933.
[7] D. Richters, M. Lass, C. Plessl, and T. D. Kühne, "A general algorithm to calculate the inverse principal p-th root of symmetric positive definite matrices," *Preprint*, Mar. 2017. [Online]. Available: https://arxiv.org/abs/1703.02456
[8] NVIDIA Corporation, "Tesla P100 data sheet," Oct. 2016.
[9] T. K. Callaway and E. E. Swartzlander, "Power-delay characteristics of cmos multipliers," in *P S Comp Arithm*, 1997, pp. 26–32.
[10] M. Ceriotti, T. D. Kühne, and M. Parrinello, "An efficient and accurate decomposition of the Fermi operator," *J Chem Phys*, vol. 129, no. 2, p. 024707, 2008.
[11] D. Richters and T. D. Kühne, "Self-consistent field theory based molecular dynamics with linear system-size scaling," *J Chem Phys*, vol. 140, no. 13, p. 134109, 2014.
[12] S. Mohr, L. Ratcliff, P. Boulanger, L. Genovese, D. Caliste, T. Deutsch, and S. Goedecker, "Daubechies wavelets for linear scaling density functional theory," *J Chem Phys*, vol. 140, no. 20, p. 204110, 2014.
[13] E. Prodan and W. Kohn, "Nearsightedness of electronic matter," *Proc Natl Acad Sci U S A*, vol. 102, no. 33, pp. 11 635–11 638, 2005.
[14] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001. [Online]. Available: http://www.scipy.org/
[15] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. Seljebotn, and K. Smith, "Cython: The best of both worlds," *Comput Sci Eng*, vol. 13, no. 2, pp. 31–39, 2011.