Do Iterative Solvers Benefit from Approximate Computing? An Evaluation Study Considering Orthogonal Approximation Methods

Michael Bromberger[1(✉)], Markus Hoffmann[1], and Robin Rehrmann[2]

[1] Computer Architecture and Parallel Processing, Karlsruhe Institute of Technology, Karlsruhe, Germany
bromberger@kit.edu
[2] Database Technology Group, Technische Universität Dresden, Dresden, Germany

Abstract. Employing algorithms of scientific computing often comes in hand with finding a trade-off between accuracy and performance. Novel parallel hardware and algorithms only slightly improve these issues due to the increasing size of the problems. While high accuracy is inevitable for most problems, there are parts in scientific computing that allow us to introduce approximation. Therefore, in this paper we give answers to the following questions: (1) Can we exploit different approximate computing strategies in scientific computing? (2) Is there a strategy to combine approaches? To answer these questions, we apply different approximation strategies to a widely used iterative solver for linear systems of equations. We show the advantages and the limits of each strategy and a way to configure a combination of strategies according to a given relative error. Combining orthogonal strategies as an overall concept gives us significant opportunities to increase the performance.

1 Introduction

Scientific computing poses a difficult challenge for people from different domains, especially in order to find a suitable trade-off between desired solution quality and computational effort. Even the high parallel capabilities of todays hardware and novel parallel algorithms do not lead to a significant reduction of these challenges because of the increasing dimensions of current problems. Hence, we rely on new ways to find suitable methods to overcome the aforementioned issues.

In recent years, the idea of an approximate computing (AC) paradigm has been gaining high attention in computer science [11]. A consideration of current applications, such as Recognition, Mining, and Synthesis (RMS) concludes that these applications have an inherent resilience against computational errors [8]. Trading off internal or external accuracy of an application allows the hardware, the programmer, or the user to improve other design goals like performance or energy consumption [3]. There already exists a wide variety of AC approaches

© Springer International Publishing AG, part of Springer Nature 2018
M. Berekovic et al. (Eds.): ARCS 2018, LNCS 10793, pp. 297–310, 2018.
https://doi.org/10.1007/978-3-319-77610-1_22

on different layers of the compute stack [11,21]. Additionally, there is quite some effort to control the degree of approximation according to given constraints [3].

In contrast, high accuracy is often inevitable for scientific computing. Hence, at first glance, it seems counterproductive to marry AC with scientific computing. However, there is already some successful work that introduces AC into scientific computing [2,17–19,22,23]. They mostly analyze the influence of data type precision on the accuracy. Asynchronous parallelization methods, which can be compared with relaxed synchronization, are well-known in numerics and show a high efficiency on GPUs [1]. But these works lack a schematic evaluation of AC on different parts inside a scientific application. Therefore, this paper is a first step to apply a holistic evaluation of AC on a widely used algorithm in scientific computing. This gives us the knowledge, where it is possible to apply AC and how we can combine orthogonal methods.

1.1 Current Status

AC approaches can be grouped according to the compute stack. Here, we order the approaches in the following:

- *Task Layer* approaches comprise skipping tasks, relaxing synchronization points [13], or exploiting approximate parallel patterns [15]. There exist run-time approaches that select a task from different approximate versions [3].
- *Algorithmic Layer* methods use the concept of loop perforation [21] or loop tiling [15]. Others rely on an automatic transformation of the code into a neural network. Sampling the input data offers a further way. Additionally, there are automatic ways to reason about the required data type.
- *Architecture Layer* approaches introduce AC into the hardware architecture. This includes neural processing units, approximated memory components [10], or entire designs that integrated dynamic accuracy and voltage scaling. Programmers can use such components through an extended ISA.
- *Hardware Layer* approaches [11] often deal with approximating processing units. This also includes providing different hardware-supported data types [6], i.e. exploit precision scaling.

Previous work shows that considering various levels and introducing different AC methods result in an enormous benefit [12]. However, such an orthogonal view is missing for scientific applications.

1.2 Methodology of the Evaluation

As previous work shows that AC can be beneficial for scientific computation, we analyze the usage of orthogonal AC methods for the Jacobi method. Firstly, we assemble representative input data for our evaluation (see Sect. 2). Then, we select suitable and promising AC approaches for our evaluation in Sect. 3. To note, we analyze the applicability and combination of orthogonal AC approaches, but we do not provide a run-time approach that controls the quality. However,

there already exist such approaches that can be used to control a combination of AC methods [11]. Our systematic evaluation compares the different approaches regarding their execution times and the relative error as described in Sect. 4. This evaluation aims to answer the following questions: How big is the influence of well-known AC methods on the accuracy of a scientific algorithm? Is it possible to combine AC methods to improve other design parameters while keeping an acceptable accuracy?

1.3 Main Findings

Based on the outcome of our experiments, the following conclusions can be drawn:

- **Conclusion 1:** There exist further AC approaches besides precision scaling which are useful for scientific computing. Loop tiling and loop truncation enable a programmer to trade-off accuracy for performance for the synchronous and parallelized Jacobi algorithm. Additionally, an approximation parameter that specifies the degree of relaxed synchronization poses an opportunity to find an optimal configuration point for accuracy and performance.
- **Conclusion 2:** Combining orthogonal AC methods leads to configuration points that cannot not be reached by a single method. Hence, this combination outperforms single methods regarding accuracy and performance. We show that coupling up to five AC methods is possible for the Jacobi method.
- **Conclusion 3:** Using a simple greedy-based algorithm, we can find suitable parameter values for the orthogonal AC methods. A user can state a desired relative error that is tolerable for the solution of the Jacobi method. Then, the algorithm finds the best possible performance for that given error by tuning the AC parameter.

2 Mathematical Background and Data Generation

A common task within scientific computing is numerically solving partial differential equations (PDEs). This is typically done by transforming the basic problem into a large scaled system of (linear) equations [9]. The finite element method, for example, transfers a weak formulation of the PDE directly into a system of linear equations:

$$Ax = b, \tag{1}$$

where x_i are the coefficients of a linear combination of basis functions for an appropriate function space, which approximate the solution of the PDE. Depending on the set of basis functions, the original problem, and the given approximation of the observed area, A has different characteristics including high dimensionality. Wisely selecting the basis functions leads to a sparse A. Hence, Krylow subspace methods are ideal candidates for solving the problem (1) [14]. Lowering the conditional number of A results in a higher convergence for those methods. This is accomplished by multiplying a suitable matrix B with A [20]. One method

to find a suitable B, the so-called preconditioning matrix, is a factorization of A based on its characteristics. A widely usable factorization is the incomplete LU-factorization [5]:

$$A \approx LU = B^{-1}, \tag{2}$$

where L and U are lower and upper triangle matrices, respectively. As for performance reasons B is embedded within the Krylow subspace method by multiplying it with a basis vector v_m of the actual Krylow subspace V_m, new systems of equations have to be computed:

$$Bv_m = y \quad \Leftrightarrow \quad LUy = v_m \quad \Leftrightarrow \quad L\tilde{y} = v_m, \quad Uy = \tilde{y}. \tag{3}$$

Because L and U are sparse but triangle matrices, typically solvers based on splitting methods like the Jacobi method are used to solve the inner systems [5].

The main challenge now is to solve these inner systems (3) very efficiently to keep the performance benefit due to fewer iterations of the Krylow subspace method. An important fact to note is that the accuracy of the solution of the inner systems only affects the convergence rate, hence it does not affect the solution of the outer method. To note, there are some important mathematical properties for solvers and preconditioning methods. First of all, the preconditioning operator B has to be invariable over the whole iteration process for most Krylow subspace methods [14]. Manipulating the updating process of the inner solver may change the operator from one iteration to another. However, methods such as FGMRES allow us to adapt the preconditioners per iteration [14].

The second problem is the convergence of the inner solver. Having a spectral radius ρ of L and U smaller than unity results in a secured convergence [4]. Although this requirement on ρ might not be fulfilled for all matrices assembled from discretization of PDEs and incomplete factorization, there are large and relevant classes of problems with resulting triangular matrices that can be solved by matrix splitting based solving methods.

Now, we take a look at the generation of the test data. The basic problem that we use is an inhomogeneous Poisson's problem with homogeneous boundary conditions on the unit square. The discretization is done with a five-point-stencil and the finite difference method. The resulting system of equations is diagonally dominant, irreducible, and can be easily scaled to any useful dimension. A is also sparse, symmetric, and positive definite. We use the Jacobi method as inner solver. The right side v_m of (3) is a set of vectors that are created as residuals within a performed CG method. To avoid misunderstandings, we would like to emphasize that we are only investigating the influence of AC on the Jacobi method. Therefore, we are only solving the resulting inner systems for evaluation purposes, but we are not trying to precondition the CG method. As mentioned before, the CG method needs an invariable preconditioning operator which is violated by our methods. Considering the influence on the preconditioning quality, for instance using FGMRES is left for future work.

3 Approximation Computing Methods

The selection of the considered approximation methods is inspired by two things. Firstly, we want to evaluate orthogonal methods which can be applied concurrently. Secondly, we decide to use approaches that seem promising and have a high standing in the approximate computing domain. Moreover, each of them have shown great success on different applications. Our selection of methods is shown in Table 1. Each of these methods offers different parameters that influence the trade-off between different design goals like accuracy and performance. We describe the meaning of each parameter in this section. Moreover, we state the useful approximation parameters.

Table 1. Overview about the considered approximation methods.

Level	Approaches	Description	Evaluation
Thread	Relaxed synchronisation	Section 3.1	Section 4.4
Data	Loop perforation, Loop tiling, and Loop skipping	Section 3.2	Section 4.3
Data type	Precision scaling and approximate memory	Section 3.3	Section 4.2
Input approximation	Input data approximation	Section 3.4	Section 4.5

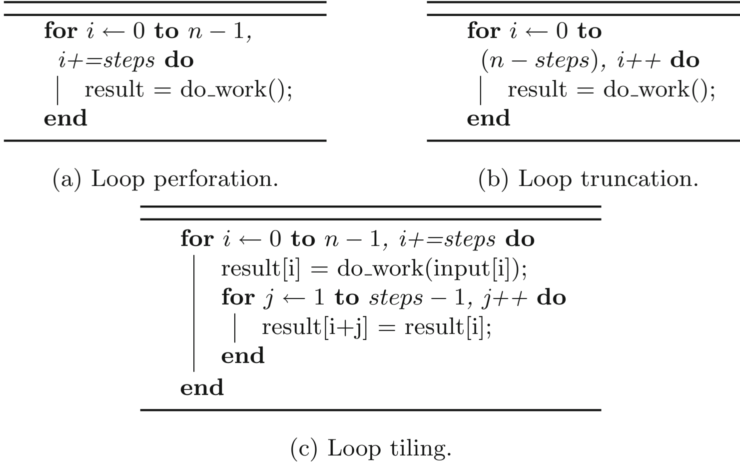
3.1 Relaxed Synchronization

Relaxed synchronization is a way to reduce the synchronization overhead introduced for a parallel execution [13]. It means that some synchronization points are intentionally violated to improve performance. However, relaxed synchronization can hamper the accuracy of the result. Hence, programmers have to take care where relaxed synchronization is viable. Barriers or synchronizations that assure to read the most recent data are good points to introduce relaxation.

For our evaluation, we use an algorithmic-specific relaxation, which are often called asynchronous methods in numerics. The used relaxation is based on a work of Anzt et al. [1]. Normally, a given starting vector is updated within each step of the Jacobi method which can be done in parallel but needs synchronization at the end of the iteration. The idea behind the relaxation is to subdivide entries of the vector in groups of a given size. Only all members of the same group are synchronized at the end of the iteration step but synchronizations between two different groups are relaxed. Anzt showed that this relaxation may lead to great speedups on GPUs. Additionally, convergence is proven for the asynchronous Jacobi method [7]. The number of groups present the approximation parameter.

3.2 Sampling

Here, we present approaches that influence the loop behavior of an algorithm. On one side, there are approaches on this level that can be considered as sampling approaches. They decide which items of the input data are used for the



(a) Loop perforation.

(b) Loop truncation.

(c) Loop tiling.

Fig. 1. Used approximation methods on the data level (sampling approaches).

computation. On the other side, we count approaches to this level that earlier stops the execution of an iterative algorithm. Figure 1 shows the schematic of these approaches.

Loop perforation (see Fig. 1a) is a well-known technique of AC on the software level [21]. The idea is to reduce the execution time of a loop by skipping iterations in between. Depending on the actual loop this essentially results in sampling the input or output. In addition, it is sometimes worth to adapt the final result, for instance using scaling for a summation of an array. Let us assume, that we only use half of the values of the sum, then multiplying the result with two can be useful. The perforation rate is the approximation parameter.

Loop truncation (see Fig. 1b) is a method that drops the last iterations of a loop. Here, the approximation parameter specifies the number of dropped iterations. Such an approach is especially useful for iterative methods. Iterative methods are commonly used in numerical mathematics. They perform a computation in such a way that they calculate a sequence of approximate solutions that ideally converge to the exact solution.

Loop tiling (see Fig. 1c) assumes that near located elements of an input have similar values [15]. Hence, it only calculates some iterations of the loop and assigns nearby outputs to the already calculated value. This actually forms a tile structure of the output. The tile size presents the approximation parameter.

3.3 On the Data Type Level

Typically, numerical algorithms rely on floating-point operations performed on the executing hardware. Many approaches in AC present designs that deal with arithmetic units, which also includes floating-point units. These approaches can be roughly grouped into two general approaches.

One deals with the precision of the operations itself [11]. This is achieved by precision scaling or by redesigning a processing unit in an approximate way. This leads to more efficient hardware designs regarding power consumption, latency, or area. The other approaches deal with approximate memory which may affect the accuracy of involved operands [10]. In general, approximate memories can lead to indeterministic stored data.

To include those approaches in our evaluation, we adapt floating-point operations within the algorithm. The first group is simulated by truncating bits of the significand (called `precision scaling`). The approximation parameter states the number of truncated bits. For the second, we introduce random bits for those less significant bits. However, this means that each memory access is affected. Therefore, we perform additional experiments, where we introduce errors according to different realistic error rates of an approximate memory [10].

3.4 Input Data Approximation

We consider a method that approximates the input data. In our test case, this can be done by taking influence to the ILU factorization as this specifies the resulting system of equations, hence the input data of the Jacobi method.

Using a sparsity pattern it is possible to specify entries of L or U that are set to zero. Therefore, the operations within the Jacobi method are reduced. The challenge is to decide which entry has the least impact on the accuracy of the Jacobi method as this is most likely the best entry to remove next.

Taking a look at the updating process of the Jacobi method it is obvious that for us the best element of L or U to remove is either the one matching to the entry of y from (3) closest to zero or the one which is closest to zero itself, both with the restriction not to remove the diagonals of the matrices. As y is unknown while computing the ILU factorization, the latter method is the one of choice. To keep the original structure of the matrices as long as possible, we additionally decide to give removing priority to the leftmost (rightmost) element of a row. Hence, it results in removing these elements first. We exploit the number of removed entries as the approximation parameter.

4 Experiments

We apply the described methods above to an iterative and parallel Jacobi solver individually. Additionally, we consider a combination of several AC methods. We run all the experiments on a AMD Opteron 6128 processor providing 64 GB of main memory. A synchronous and parallel version of the Jacobi solver executed using 32 threads is our base line. We parallelize over matrix rows. The parallel algorithm requires 130.1 ms for a matrix dimension of 1024^2 and 631.2 ms for a dimension of 2048^2. If not otherwise mentioned, we set the iteration count to 10. Stopping the iterative method after 10 iterations results in a relative error of roughly 10^{-4} compared to the exact solution independent from the matrix dimension d.

4.1 Evaluation Metrics

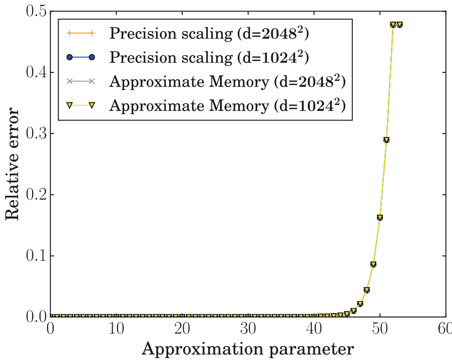
For the accuracy, we calculate the relative error

$$E_{rel} = \frac{||\boldsymbol{x} - \tilde{\boldsymbol{x}}||_2}{||\boldsymbol{x}||_2},$$

where $\boldsymbol{x}$ is the solution vector of the base line and $\tilde{\boldsymbol{x}}$ the solution of the approximate version. Moreover, we measure the performance stated as execution time if possible. In other cases, we include realistic numbers from the literature.

4.2 Influence of Approximate Computing on the Data Type Level

In this section, we investigate how the internal data type precision impacts the accuracy of the solution vector. Since we cannot perform these experiments on current hardware, we use an emulation scheme to evaluate the influence of precision. The reason is that current hardware does not provide other floating-point data types apart from float or double in general. We consider two well-known AC methods: precision scaling and approximate memory. Figure 2a shows the impact of these methods on the relative error. We vary the number of influenced precision bits of the significands from 53 to 0. We can see that for the given linear system, the most of the least significant bits of the significand play a minor role for the accuracy. Moreover, the results are more or less independent from the matrix dimension d and the way how we influence the data type precision. 13 bits are enough to have almost no additional error compared to the base line. Having less than roughly 8 correct bits leads to an exponential increase in the relative error. However, according to literature it is not very likely that all memory reads are affected by approximation. It actually depends on how this approximation method is implemented. A common way is to increase the refresh cycle time of a DDR memory bank, which can significantly save energy. Depending on this



(a) Precision scaling.

Error rate	Relative error $d = 1024^2$	$d = 2048^2$
1.3×10^{-4}	0.00234583	0.0026825
2.0×10^{-5}	0.00221096	0.0010068
3.8×10^{-6}	0	0
2.6×10^{-7}	0	0

(b) Approximate Memory.

Fig. 2. Influence of the data type precision on the accuracy.

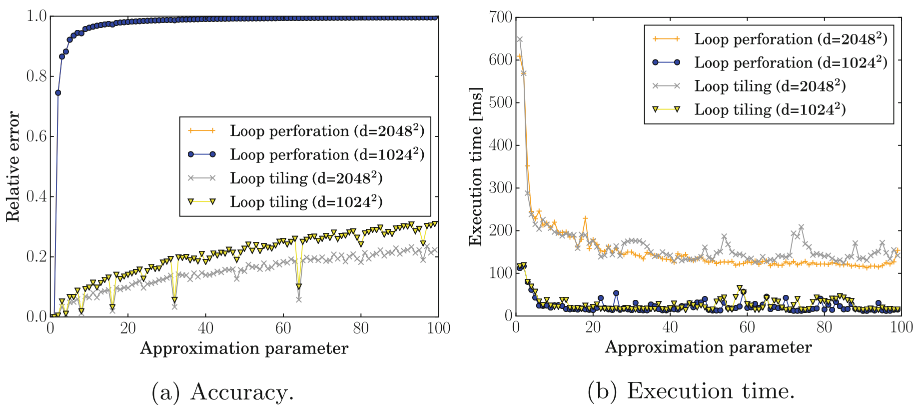
increase the error rate of getting wrong results from the memory also raises. For some realistic values, we consider how this error rate impacts the accuracy of the Jacobi solver, see Fig. 2b. Even if we have relatively high error rates, for instance 1.3×10^{-4}, the influence on the accuracy is not drastic. Such an approximate memory approach decreases the power required for refresh up to 25% having an error rate of 1.3×10^{-4} [10]. Getting the actual performance or energy gain is very difficult, since it would require to build such a hardware and to evaluate the wanted metrics. Here, we show the potential of the reduction in precision bits.

4.3 Analysis of Approximate Computing Loop Strategies

A common method in AC is to adapt the execution of iterations for a loop. This essentially leads to skipping iterations or a sampling scheme on the input data. Figure 3 shows the impact of loop perforation and loop tiling for different approximation parameters (called `steps` in Fig. 1). The method `loop perforation` is not applicable at all for the considered algorithm, since the error exponentially increases with the approximation parameter. In contrast, `loop tiling` works quite well. Especially, small values for the approximation parameter still lead to small errors. We can see an influence of the dimension on the accuracy for `loop tiling`. A smaller dimension shows a higher error behavior.

Fortunately, the execution time significantly decreases for small parameter values. Larger values have no further considerable benefit regarding the execution time. The rationale behind is that at a certain point the synchronization overhead of the parallelization and other parts of the algorithm, where the AC methods have no effect, have the main impact on the execution time.

`loop truncation` is a natural way to approximate iterative methods. It just stops the iterative method before it converges. Figure 4 shows the accuracy and



(a) Accuracy.

(b) Execution time.

Fig. 3. Influence of loop perforation and loop tiling (Measurements are overlapping for loop perforation).

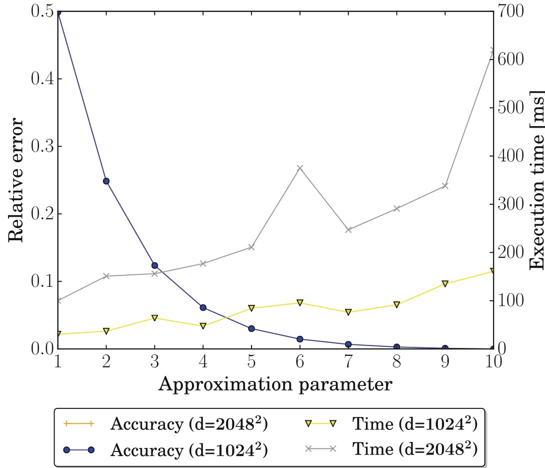


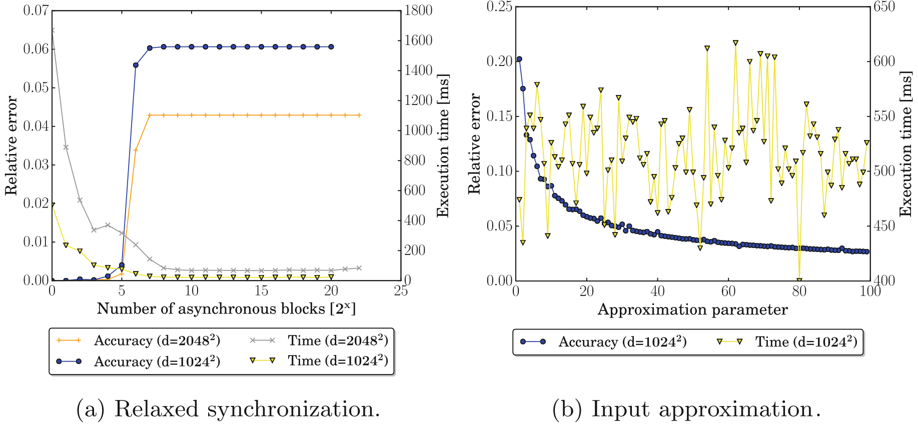
Fig. 4. Influence of loop truncation regarding accuracy and performance (Measurements for accuracy are overlapping).

execution time for different stop points. A stop point specifies the number of allowed iterations. Again the relative error is almost independent from the matrix dimension. The error exponentially decreases with the iterations at the beginning and then requires some time to converge. The execution time for large dimensions scales roughly linearly with the number of iterations. For small dimensions, the synchronization overhead is quite high.

To sum up, `loop perforation` is not a useful approach for the Jacobi method. Regarding the error and performance, `loop truncation` provides the best solution in general. However, `loop tiling` can be a useful method for larger allowed relative errors.

4.4 Accuracy Degradation Caused by Relaxed Synchronization

In the following experiment (see Fig. 5a), we investigate the influence of relaxed synchronization on the accuracy of the result vector. A higher number of blocks states that more synchronizations are relaxed during the execution. The relaxation method introduces a small error until the number of blocks is larger than the number of available cores, in our case $2^5 = 32$. At this point, we can see a high increase of the relative error. In contrast, the optimal point regarding performance is reached when the number of blocks is roughly eight times the number of cores. The curves show similar behavior for different matrix dimensions, but the relative error is smaller for the larger dimensions. The performance gain is more significant for larger matrix dimensions.



(a) Relaxed synchronization.

(b) Input approximation.

Fig. 5. Consideration of relaxed synchronization and input approximation on the Jacobi method (We are aware of the strange time measurements but unfortunately it is unclear where the oscillation comes from. However, they are reproduceable).

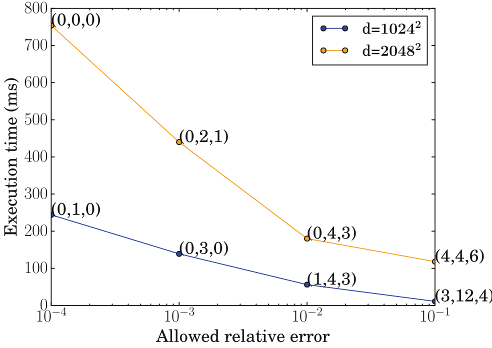
4.5 Input Approximation

Instead of using approximation in the algorithm itself, one can adapt the input data. Therefore, we remove certain inputs according to a method described in Sect. 3.4. The approximation parameter presents an offset which specifies the rows of the input matrix that will be affected. For instance, 20 means that we influence each 20th row. In general, affecting fewer rows leads to a reduction of the error. Until a parameter value of 20, this reduction is exponential (see Fig. 5b). Afterwards, the error decreases slowly.

However, we cannot see that removing certain inputs have a clear influence on the execution time. There are strong variations in the execution time which means that they are independent from the approximation parameter. According to these results, we draw the conclusion that input approximation is not useful for our test case.

4.6 Putting Everything Together

Now, we are able to combine multiple and orthogonal AC methods. According to the results so far, we include `loop truncation`, `loop tiling`, relaxed synchronization and precision scaling. All of them have an approximation parameter that can be tuned. We set these approximation parameter values according to a given relative error, which represents our constraint. To find a good configuration of parameter values that satisfies these constraints, we exploit a known greedy algorithm [16] based on steepest ascent hill climbing. For the first test, we exclude precision scaling, since we cannot make performance measurements for this method. Then, the task of the greedy algorithm is to find the parameter values which offers the best performance under the given error constraint. We adapt



(a) Three approximation methods.

max E_{rel}	Parameter set	
	$d = 1024^2$	$d = 2048^2$
1×10^{-4}	$(0 \mid 2^1 \mid 0 \mid 32)$	$(0 \mid 2^1 \mid 0 \mid 52)$
1×10^{-3}	$(0 \mid 2^2 \mid 1 \mid 35)$	$(0 \mid 2^3 \mid 0 \mid 52)$
1×10^{-2}	$(0 \mid 2^4 \mid 3 \mid 40)$	$(1 \mid 2^4 \mid 3 \mid 52)$
1×10^{-1}	$(4 \mid 2^4 \mid 6 \mid 52)$	$(2 \mid 2^{12} \mid 2 \mid 52)$

(b) Four approximation methods.

Fig. 6. Considering multiple orthogonal approximation methods for the Jacobi method. Parameter set $(TI|RS|TR|PS)$ TI: loop tiling, RS: relaxed synchronization, TR: loop truncation, PS: precision scaling

approximation parameters in a way that higher values present a more aggressive approximation level. The results are shown for different error constraints in Fig. 6a. As we can see, the configuration algorithm tunes the parameter of all three orthogonal methods. Hence, the combination of methods is beneficial to reach good performance points for different error constraints. Allowing a relative error of 1%, we get a performance improvement of roughly 300% compared to the 32 threaded basis version. Moreover, a 10% allowed error leads to almost a speed-up of 6.

For the found configuration points, we further consider the potential of precision scaling, see Fig. 6b. All configurations enable us to further introduce AC on the data type level. This allows a hardware designer to approximate hardware arithmetic units for the algorithm under test. Additionally, another possibility is to include approximate DRAM according to Sect. 4.2 as fifth parameter.

4.7 Discussion

Taking a look on our results, we see that not only a single AC strategy can be useful in terms of scientific computing, but also a combination of strategies, especially in the context of preconditioning, where high accuracy is unnecessary in most cases. Moreover, it is possible to estimate tolerable computing errors. Hence, we are sure, that it is possible to reduce computation times for the inner solver dramatically by reducing accuracy to a reasonable degree. Of course, we are aware that accompanying quality loss of the preconditioning method can result in lower convergence rates for the Krylow subspace method. But the results of the combined AC strategies show that remarkable speed-ups can be gained with careful accuracy reductions.

Based on our results we want to use a flexible Krylow subspace method, like FGMRES, in combination with a set of AC strategies for the preconditioning method adjusted with a tunable accuracy parameter. Although we have not measured the quality of the preconditioning method yet, we think that this setting will lead to great speed-ups for the whole preconditioned solver. Additionally, further AC strategies, like reformulating the ILU solver into an iterative method and skipping iterations, which does not influence the speed of the Jacobi method but the quality of the preconditioning method, can be added easily.

5 Conclusion and Future Directions

In this paper, we considered orthogonal approximate computing (AC) methods and how they influence the accuracy and performance trade-off of a scientific computing algorithm. All methods were experimentally investigated for the Jacobi method performing on realistic data. Hence, we applied the first extensive, holistic, and schematic evaluation of AC on a scientific algorithm. While single methods already can be seen as useful, a combination of them results in a much higher gain. For instance, allowing 1% relative error we achieve an acceleration of 3 compared to the parallel version of Jacobi (32 threads).

For future work it is mandatory to extend the test setting to the complete Krylow subspace method to measure the effects of AC methods on the quality of the preconditioning. With this enlarged setting, the usefulness of the presented methods can be considered in a broader spectrum.

References

1. Anzt, H., Chow, E., Dongarra, J.: Iterative sparse triangular solves for preconditioning. In: Träff, J.L., Hunold, S., Versaci, F. (eds.) Euro-Par 2015. LNCS, vol. 9233, pp. 650–661. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48096-0_50
2. Anzt, H., Dongarra, J., Quintana-Ortí, E.S.: Adaptive precision solvers for sparse linear systems. In: Proceedings of the 3rd International Workshop on Energy Efficient Supercomputing, p. 2. ACM (2015)
3. Baek, W., Chilimbi, T.: Green: a framework for supporting energy-conscious programming using controlled approximation. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (2010)
4. Bagnara, R.: A unified proof for the convergence of Jacobi and Gauss Seidel methods. SIAM Rev. **37**, 93–97 (1995)
5. Benzi, M.: Preconditioning techniques for large linear systems: a survey. J. Comput. Phys. **182**, 418–477 (2002)
6. Bromberger, M., Heuveline, V., Karl, W.: Reducing energy consumption of data transfers using runtime data type conversion. In: Hannig, F., Cardoso, J.M.P., Pionteck, T., Fey, D., Schröder-Preikschat, W., Teich, J. (eds.) ARCS 2016. LNCS, vol. 9637, pp. 239–250. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30695-7_18
7. Chazan, D., Miranker, W.: Chaotic relaxation. Linear Algebra Appl. **2**, 199–222 (1969)

8. Chippa, V., Chakradhar, S., Roy, K., Raghunathan, A.: Analysis and characterization of inherent application resilience for approximate computing. In: Proceedings of the 50th Annual Design Automation Conference, DAC 2013, pp. 113:1–113:9. ACM, New York (2013)
9. Larsson, S., Thomee, V.: Partial Differential Equations with Numerical Methods. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-88706-5
10. Liu, S., Pattabiraman, K., Moscibroda, T., Zorn, B.G.: Flikker: saving DRAM refresh-power through critical data partitioning. ACM SIGPLAN Not. **47**(4), 213–224 (2012)
11. Mittal, S.: A survey of techniques for approximate computing. ACM Comput. Surv. (CSUR) **48**, 62:1–62:33 (2016)
12. Raha, A., Venkataramani, S., Raghunathan, V., Raghunathan, A.: Energy-efficient reduce-and-rank using input-adaptive approximations. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. **25**(2), 462–475 (2017)
13. Renganarayana, L., Srinivasan, V., Nair, R., Prener, D.: Programming with relaxed synchronization. In: Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability, pp. 41–50. ACM (2012)
14. Saad, Y.: Iterative Methods for Sparse Linear Systems. PWS, Boston (1996)
15. Samadi, M., Jamshidi, D.A., Lee, J., Mahlke, S.: Paraprox: pattern-based approximation for data parallel applications. ACM SIGARCH Comput. Archit. News **42**, 35–50 (2014)
16. Samadi, M., Lee, J., Jamshidi, D.A., Hormati, A., Mahlke, S.: SAGE: self-tuning approximation for graphics engines. In: Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 13–24. ACM (2013)
17. Schaffner, M., Gurkaynak, F.K., Smolic, A., Kaeslin, H., Benini, L.: An approximate computing technique for reducing the complexity of a direct-solver for sparse linear systems in real-time video processing. In: 2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC), pp. 1–6. IEEE (2014)
18. Schöll, A., Braun, C., Wunderlich, H.J.: Applying efficient fault tolerance to enable the preconditioned conjugate gradient solver on approximate computing hardware. In: 2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), pp. 21–26. IEEE (2016)
19. Schöll, A., Braun, C., Wunderlich, H.J.: Energy-efficient and error-resilient iterative solvers for approximate computing. In: Proceedings of the 23rd IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS 2017), pp. 237–239 (2017)
20. Shewchuk, J.R.: An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. School of Computer Science, Carnegie Mellon University, Pittsburgh, August 1994
21. Sidiroglou-Douskos, S., Misailovic, S., Hoffmann, H., Rinard, M.: Managing performance vs. accuracy trade-offs with loop perforation. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE 2011, pp. 124–134. ACM, New York (2011)
22. Zhang, Q., Tian, Y., Wang, T., Yuan, F., Xu, Q.: Approxeigen: an approximate computing technique for large-scale eigen-decomposition. In: Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, pp. 824–830. IEEE Press (2015)
23. Zhang, Q., Yuan, F., Ye, R., Xu, Q.: Approxit: an approximate computing framework for iterative methods. In: Proceedings of the 51st Annual Design Automation Conference, pp. 1–6. ACM (2014)