Improving the Performance of Task-based Linear Algebra Software with Autotuning Techniques on Heterogeneous Architectures

Jesús Cámara[1][0000−0003−2176−8273], Javier Cuenca[2][0000−0002−8763−756X], and Murilo Boratto[3][0000−0003−3908−4451]

[1] Dept. of Informatics, University of Valladolid, Spain
jesus.camara@infor.uva.es
[2] Dept. of Engineering and Technology of Computers, University of Murcia, Spain
jcuenca@um.es
[3] Dept. of Sciences, University of Bahia, Brazil
muriloboratto@uneb.br

Abstract. This work presents several self-optimization strategies to improve the performance of task-based linear algebra software on heterogeneous systems. The study focuses on Chameleon, a task-based dense linear algebra software whose routines are computed using a tile-based algorithmic scheme and executed in the available computing resources of the system using a scheduler which dynamically handles data dependencies among the basic computational kernels of each linear algebra routine. The proposed strategies are applied to select the best values for the parameters that affect the performance of the routines, such as the tile size or the scheduling policy, among others. Also, parallel optimized implementations provided by existing linear algebra libraries, such as Intel MKL (on multicore CPU) or cuBLAS (on GPU) are used to execute each of the computational kernels of the routines. Results obtained on a heterogeneous system composed of several multicore and multiGPU are satisfactory, with performances close to the experimental optimum.

Keywords: autotuning · linear algebra · heterogeneous computing · task-based scheduling.

1 Introduction

In recent years, the increasing heterogeneity of computational systems has made the efficient execution of scientific applications a major challenge. Tipically, these systems consist of a set of compute nodes made up of several parallel devices, such as multicore processors, graphics accelerators, or programmable devices, with a different number of processing elements and memory hierarchy levels each. Thus, efficiently exploiting the computational capacity of the whole system is not an easy task, and even more so in parallel. In many cases, traditional numerical algorithms need to be redesigned to perform the computations parallel and efficiently. The advances in the development of highly optimized

libraries have made it possible to tackle this problem, improving the overall performance of the routines involved. In numerical dense linear algebra, several libraries have been developed for specific parallel devices. The most commonly used are PLASMA [7] or Intel MKL [10] for multicore processors, cuBLAS [13] for NVIDIA GPUs or MAGMA [15] for multicore+GPU systems. Depending on the problem to be solved, these libraries can be used in an isolated way (for a specific device) or combined together when the workload is distributed among the different parallel computing devices. Also, some of them are designed to better exploit the different levels of parallelism of the architecture by using vectorization or multi-threading features (such as Intel MKL) and others, such as PLASMA, focus on using fine-grained parallelism to execute operations in parallel, representing them as a sequence of small tasks that can be dynamically scheduled. Regardless of the purpose of the library, they all provide reasonably good performance without taking into account the values for a set of adjustable parameters related to the hardware platform or the library routines themselves. However, in order to exploit the whole system efficiently, it is also necessary to search for the best values for these parameters.

In the literature, two general researching lines can be found that aim at this objective. On the one hand, there are techniques based on an empirical search supported heuristically by a certain knowledge of the platform architecture. This autotuning approach was introduced by ATLAS [17] and its predecessor PHiPAC [6]. On the other hand, code that is competitive with implementations generated via empirical search could also be produced by analytical proposals [11], either modelling the behaviour of the hardware when the software is executed [12] or, at higher level, modelling the performance of the software when it is running on this hardware [8]. In addition, the emergence of heterogeneous systems integrating accelerator devices with a wide variety of parallel architectures, such as GPUs, has led to the development of proposals focused on how to optimize linear algebra kernels for such platforms [4]. Also, hybrid methods can also be found, such as applying a hierarchical autotuning method at both hardware and software levels [9]. Similarly, wizard frameworks for developers have emerged, such as BOAST [16], which aim to minimise the effort of migrating high-performance computing kernels between platforms of different nature by providing an embedded domain-specific language to describe the kernels and their possible optimization.

The working scenario differs when the software includes a set of components that work together to execute the routines. This is the case for task-based libraries, which are of interest in the field of high-performance computing. This work focuses on Chameleon [1], a task-based dense linear algebra software for heterogeneous architectures. This library is based on the Sequential Task Flow (STF) paradigm and runs on top of a runtime system, which dynamically distributes and executes the tasks (basic computational kernels) among the available computing resources (multicore or GPUs) using a Directed Acyclic Graph (DAG) of tasks. Since the execution of tasks is handled by the runtime system, the self-optimization strategies proposed to select the best values for the

adjustable parameters should be extended to consider, in addition, the configurable scheduling parameters.

In previous approaches used to tune the Chameleon library [2], the values of several adjustable parameters (block size and scheduling policy) were selected to get the best asymptotic performance for the whole set of problem sizes. In a preliminary comparative study carried out with the Cholesky routine on multicore CPU and CPU+multiGPU systems, the best overall values selected for these adjustable parameters have been $\{nb = 512, sched = \text{“eager”}\}$ and $\{nb = 576, sched = \text{“dmdas”}\}$, respectively. However, the performance of the routine improves when the values are properly selected for each problem size, as shown in Fig. 1. By default, Chameleon considers a fixed value for these adjustable parameters regardless of the problem size, which leads to a loss in performance as the problem size increases. With the asymptotic values, instead, there is a loss in performance for small problem sizes. Therefore, these results demonstrate the importance of having a self-optimization process that allows to overcome the performance shortcomings offered by both approaches.
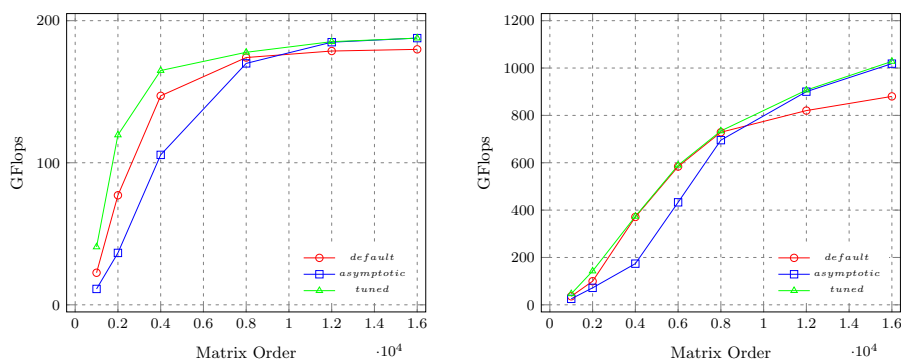

Fig. 1. Performance obtained on multicore CPU (left) and CPU+multiGPU (right) systems with the Cholesky routine of Chameleon when using the default values for the adjustable parameters (`default`), the overall values (`asymptotic`) and the best values selected (`tuned`) with the self-optimization methodology.

These first pieces of evidence together with other preliminary results [3] have led to undertake this work of developing a collaborative framework that jointly addresses two self-optimization methodologies: on the one hand, at installation time by using a training engine and, on the other hand, at execution time through the use of the dynamic task-based scheduling provided by Chameleon. The training engine can use different search strategies depending on the number and type of the adjustable parameters: routine parameters (block size and inner block size), system parameters (number and type of computing resources) and scheduling parameters (policies used by the runtime system). With this approach, at installation time, a performance map of the routine is obtained and stored in a

database. After that, at execution time, this information is used to quickly make better decisions for any specific problem to be solved.

The rest of the paper is organized as follows. Section 2 shows the general scheme of execution of a linear algebra routine in Chameleon. In Section 3, the self-optimization methodology for selecting the best values of a set of adjustable parameters when working with task-based libraries is presented. Section 4 shows the experimental study carried out with different routines of Chameleon when applying the proposed methodology. Finally, Section 5 concludes the paper and outlines future research lines.

2 The Chameleon Library

As mentioned, this paper focuses on Chameleon, a task-based dense linear algebra software that internally uses StarPU [5], a runtime system to dynamically manage the execution of the different computational kernels on the existing hybrid computing resources.

Fig. 2 shows the steps to execute a linear algebra routine of Chameleon using the StarPU runtime system. As Chameleon is derived from PLASMA, each routine is computed by following a tile-based algorithm. First, the tasks to compute each of the data tiles in the matrix layout are created based on the STF paradigm and the DAG is generated with dependencies between them. These tasks correspond to the basic algebraic operations (called kernels) involved in the computation of the routine and act on different blocks of data, whose size depends on the value specified for the tile size (nb in the Figure). Then, the tasks are scheduled using one of the scheduling policies provided by StarPU and executed on the available computing resources using efficient implementations of the basic kernels, such as those provided by the Intel MKL and cuBLAS libraries.
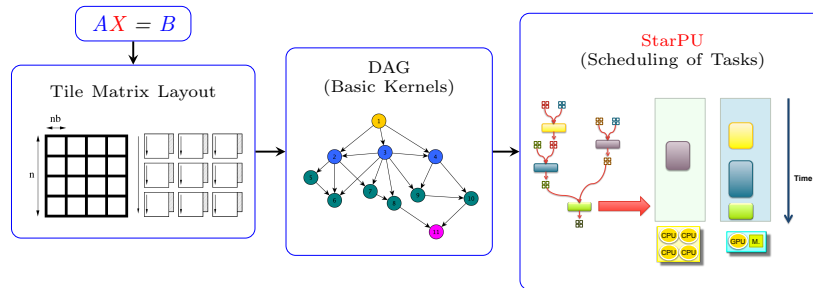

Fig. 2. Execution of a linear algebra operation in Chameleon using StarPU.

3   Self-Optimization Methodology

This section describes the proposed self-optimization methodology for selecting the best values for the adjustable parameters that affect the performance of the routines in task-based libraries. Fig. 3 shows the general operation scheme. It consists of three main phases: selection of the search strategy, training of the routine with the selected strategy for different problem sizes and values for the set of parameters, and evaluation of the performance obtained by the routine using a different set of problem sizes.
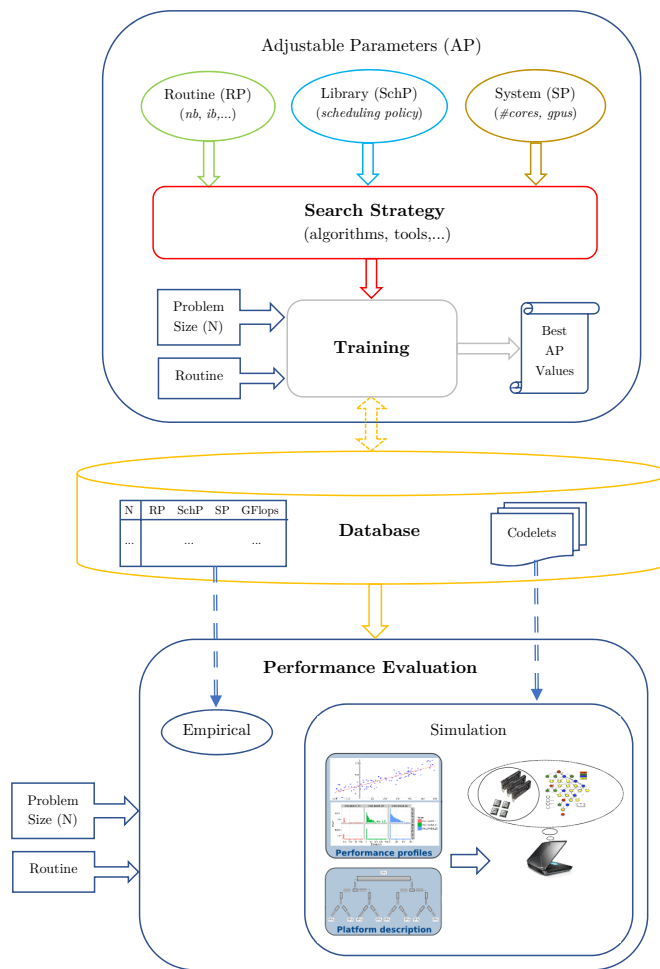

Fig. 3. Self-optimization methodology for task-based linear algebra libraries.

3.1   Selecting the Search Strategy

In task-based libraries, the performance of the routines depends mainly on the values selected for three subsets of adjustable parameters, AP:

- Routines Parameters, RP: block size and inner block size.
- System Parameters, SP: number and type of computing resources.
- Scheduling Parameters, $SchP$: scheduling policy used by the runtime system.

In order to find the best values for these AP, it is necessary to make use of search strategies. Nevertheless, the strategy chosen will be determined by the number of AP considered and the way it is conducted (exhaustive or guided). The proposed self-optimization methodology includes several search strategies to select the best values for RP:

- **Exhaustive_NB**: the routine is executed by varying the block size, nb, for each problem size, n. As a result, the execution time and the performance obtained for each (n, nb) is stored.
- **Exhaustive_NB+IB**: the routine is executed by varying the block size, nb, for each problem size, n, by increasing the value of the inner block size, ib, until it reaches the value of the current block size. As a result, the execution time and the performance obtained for each (n, nb, ib) is stored.
- **Guided_NB**: the routine is executed starting with the first problem size and the first block size considered, nb_1, whose value is increased until the performance decreases. The process continues with the next problem size (in ascending order), taking as starting point the value of nb for which the best performance was obtained for the previous problem size. Then, a bidirectional search is performed on the nb value (with increasing and decreasing values) until the performance obtained worsens the one obtained so far. The process then continues with the remaining problem sizes. As a result, for each problem size, the execution time and the performance obtained with the best value for nb is stored.
- **Guided_NB+IB_2D**: the routine is executed starting with the first problem size and the first block size considered, (n_1, nb_1), and the inner block size, ib, is increased until the value of nb_1 is reached or the performance obtained worsens the one obtained so far. To prevent the process from falling into a local minimum, a percentage value is used as tolerance to decide if the search continues to the next value of ib. The process then continues with the next value of nb for the same problem size, taking as starting point the ib value with which the best performance was obtained for the previous block size. Then, a bidirectional search is performed on the ib value (with increasing and decreasing values) using the percentage value considered until the performance decreases or the value of nb is reached. Once the search has finished for the last value of nb, the execution time and the performance obtained with the best pair (nb, ib) for the current problem size is stored. Next, the following problem size is selected (in ascending order) and the search process is repeated for ib, starting with the first block size, nb_1, and the best ib obtained for the previous problem size.

- **Guided_NB_1D+IB_2D**: the routine is executed in a similar way to the previous strategy, but when the process moves to the next problem size, it uses as starting point the best (nb, ib) values obtained for the previous problem size. The percentage value is also used to prevent the process from stalling at a local minimum, but only when varying the inner block size, ib.
- **Guided_NB_2D+IB_2D**: the routine is executed in a similar way to the previous strategy, but when the process moves to the next problem size, the bi-directional search is applied both in the block size, nb, and the inner block size, ib, taking as starting point the best (nb, ib) values obtained for the previous problem size. Again, the percentage value is used, but in this case for both the bidirectional search of nb and ib.

In addition, the system parameters (SP) can be considered. By default, the Chameleon library only uses the CPU cores available on the system. Thus, a non-expert user might consider using all the computing resources in the system. However, this does not always offer the best performance. Therefore, a search strategy should be used to select the appropriate number of GPUs and CPU cores to use for each problem size. The proposed strategy consists of adding computing resources in increasing order of computational capacity by applying at each step a guided strategy to select the best values of the routine parameters (RP) for the first problem size considered. Then, the process continues with the next problem size, using as a starting point the best values obtained for the RP for the previous problem size and keeps adding new computing resources as long as the performance does not worsen the one obtained so far.

The same idea can be applied to select the best values for the scheduling parameters ($SchP$). In this case, the parameter to consider in the Chameleon library is the scheduling policy used by the runtime system. Therefore, the search strategy will consist of selecting the best one among those offered by StarPU. This strategy can be used in conjunction with the previous one if the best values for all AP have to be obtained.

3.2 Training the Routines

In the training phase, the routines are executed in the system by varying the values of the AP for each input problem size according to the selected search strategy. This phase is run only once for each routine and set of problem sizes. As a result, the selected values for the AP together with the problem size, n, and the performance obtained (in million floating point operations per second) are stored in a database for further use. As will be shown in the experiments, guided strategies will considerably reduce the training time of the routines compared to the use of exhaustive ones.

3.3 Validating the Methodology

Once the routines have been trained with a set of problem sizes and varying the values for the AP, the proposed methodology can then be validated. To do so,

a validation set of problem sizes is used. As shown in Fig. 3, both an empirical and a simulated approach can be considered:

- Empirical: the routine is executed for each problem size, n, of the validation set using the best AP values stored in the database for the problem size closest to n.
- Simulated: for each problem size, n, both the AP values and the *codelets* of the computational kernels involved in the routine are obtained from the database to be used by SimGrid [14] to predict the performance. The *codelets* are automatically created for each basic kernel during the training phase and store performance data related with the execution of each basic kernel for the values used for the problem size and the routine parameters.

Finally, to check the goodness of the decisions taken (AP values), the performances obtained are compared with those that would have been obtained if an exhaustive search for the best AP values would have been carried out directly for each problem size in the validation set.

4   Experimental Study

The experimental study focuses mainly on two representative Chameleon routines: the LU and QR factorizations, although some results will also be shown for the Cholesky routine. We consider the LU routine without pivoting because the one with pivoting is not available in Chameleon as the cost of finding the pivot and executing the swap cannot be efficiently handled. Since both the LU and QR routines are implemented by following a tile-based algorithmic scheme, the impact of the block size on performance will be analyzed. In addition, the inner block size will be also considered, since LU and QR routines uses it to perform the matrix factorization.

The experiments are conducted with the search strategies described in Sec. 3.1 to select the best values of the different adjustable parameters (RP, SP, $SchP$). First, the RP values are selected. Next, the SP values and finally, the $SchP$ values. Training times are also analyzed and an example of validation for the proposed methodology is shown.

The heterogeneous platform used consists of five hybrid computing nodes, but the experiments have been performed on **jupiter**, the most heterogeneous one, which is composed of 2 Intel Xeon hexa-core CPUs (12 cores) and 6 NVIDIA GPUs (4 GeForce GTX590 and 2 Tesla C2075). Also, the Intel MKL and the cuBLAS library are used to run the basic computational kernels scheduled by the runtime system on multicore and GPUs, respectively.

In the experiments, the following set of problem sizes {2000, 4000, 8000, 12000, 16000, 20000, 24000, 28000} and block sizes {208, 256, 288, 320, 384, 448, 512, 576} are used. The values chosen for the block sizes are a representative subset of those that allow to improve the performance of the `gemm` kernel for the problem sizes considered. This kernel covers most of the computing time of the routines under study, therefore, the impact of the block sizes selected

for this kernel will have an impact on the overall performance of the routines. Nevertheless, the study could have been carried out with any other set of values.

4.1   Selecting the Routine Parameters

As shown in Fig. 3, the selection of the best values for the RP is carried out during the training phase and then stored in the database. Depending on the search strategy used, the values selected for the RP may differ, but also the time needed to obtain these values. The study will focus on selecting the best values for the block size and the inner block size in the LU and QR factorization routines by applying the search strategies described in Sec. 3.1.

First, the study is carried out with the LU factorization routine. Fig. 4 (top left) shows the training time (in seconds) for each problem size when using the Exhaustive_NB+IB strategy, with a total of 6 hours and 10 minutes. Fig. 4 (top right), on the other hand, shows the pair (nb, ib) with which the best performance is obtained for each problem size. It is noted that for problem sizes above 8000, the best value for nb always corresponds to the largest value considered in the set of block sizes. In contrast, the values selected for ib are small and vary between 16 and 56 depending on the selected block size.

As shown, the exhaustive search strategy provides good performances, but the training time is high. To reduce it, the guided search strategies (Guided_NB+IB_2D and Guided_NB_2D+IB_2D) are used to select the best values for nb and ib. Fig. 4 (middle left) shows the training time (in seconds) spent for each problem size, and Fig. 4 (middle right) shows the performance obtained with the (nb, ib) values selected by the Guided_NB+IB_2D search strategy. In this case, the total time spent is 32 minutes. Similarly, Fig. 4 (bottom) shows, respectively, the training time (in seconds) and the performance obtained with the Guided_NB_2D+IB_2D strategy for each problem size. Now, the training time spent is only 6 minutes, less than the required by the Guided_NB+IB_2D strategy and much less than the 6 hours required by the Exhaustive_NB+IB.

Next, the same study is carried out with the QR factorization routine for the same set of problem and block sizes. Fig. 5 (top left) shows the training time (in seconds) for each problem size when using the Exhaustive_NB+IB strategy. The time is about 29 hours and 57 minutes, which is longer than in the LU routine because the LU factorization does not perform pivoting (as indicated before). Fig. 5 (top right) shows the performance obtained for each problem size when using the best (nb, ib) values. Again, for problem sizes above 8000, the best value for the block size, nb, always corresponds to the largest value considered in the set of block sizes. However, the values selected for ib are sligthly higher than the ones selected for the LU routine, varying from 72 to 160. Next, the Guided_NB+IB_2D and Guided_NB_1D+IB_2D search strategies are applied to reduce the training time. Fig. 5 (middle left) shows the time spent (in seconds) for each problem size, and Fig. 5 (middle right) shows the results obtained when using the Guided_NB+IB_2D strategy. In this case, the time spent is 8 hours and 24 minutes, 64% less than the exhaustive one. Similarly, Fig. 5

(bottom left) shows the training time (in seconds) for each problem size, and Fig. 5 (bottom right) shows the performance and the (nb, ib) values obtained for each problem size when using the Guided_NB_1D+IB_2D strategy. Now, the time spent is 87 minutes, 86% and 97% less than using the Guided_NB+IB_2D and Exhaustive_NB+IB_2D strategies, respectively.

Finally, Fig. 6 shows a comparison of the performance results obtained for the LU and QR routines when using the exhaustive and guided search strategies. In Fig. 6 (left) it is observed that the performance obtained by the LU routine with the guided search strategies overlaps with that obtained using the exhaustive search. A similar behaviour is observed in Fig. 6 (right) for the QR routine, especially when the Guided_NB_1D+IB_2D strategy is used and the problem size increases. Therefore, the use of guided strategies will allow to obtain satisfactory performance results with the selected RP values and with reasonable training times.

4.2   Selecting the System and Scheduling Parameters

Besides the routine parameters, there are a set of AP whose values may be selected during the training phase to further improve the performance. In task-based libraries, such as Chameleon, the number and type of computing resources to use and the scheduling policy used by the dynamic task scheduler can be specified when executing a routine. Since the experiments are performed on a heterogeneous node, both the number of CPU cores and GPUs will be considered. In addition, as the execution of the computational kernels of the routines is handled at run-time by StarPU, which decides how to manage the execution of the kernels on the different computing resources of the system, different scheduling policies are considered. Some of them (such as dm, $dmda$ or $dmdas$) use the information from the *codelets* generated during the training phase. Other policies, however, are only based on priorities (such as *eager* and *random*), the load of task queues (such as *ws* and *lws*) or the availability of the computing resources. Therefore, the proposed methodology could be also applied to select the best values for the system parameters as well as the best scheduling policy among all those offered by StarPU.

Table 1 shows the values obtained for the Cholesky routine for a set of problem sizes. The GPU IDs are displayed to show the order in which they are chosen by the search strategy described in 3.1, as the scheduler assigns tasks to computing resources following a scheme based on priorities and dependencies between the data required by the different computational kernels, but it does not take into account the computational capacity of these resources. The results show how an appropriate selection of all the adjustable parameters (block size, number of CPU cores and number of GPUs, and scheduling policy) allows to obtain a performance improvement between 10% and 50% with respect to that obtained by the Chameleon library with the default running configuration. Also, the selected values differ for most problem sizes, hence the importance of using a self-optimization methodology that allows to select the best AP values for each problem size to improve the performance of the routines.
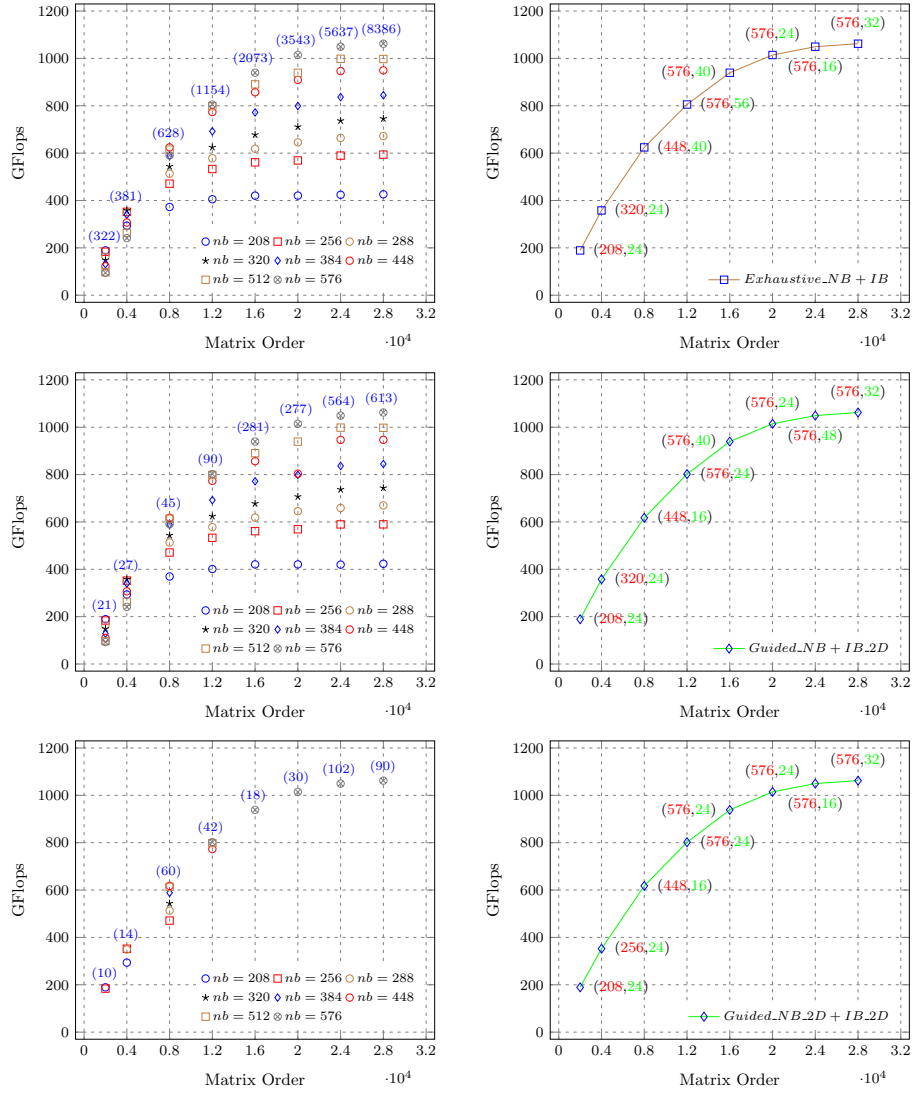
Fig. 4. LU factorization. Training time (left, in seconds) for each problem size when using the exhaustive (top left) and guided (middle and bottom) search strategies, and performance obtained (right) with each strategy when using the best (*nb*, *ib*) values for each problem size.
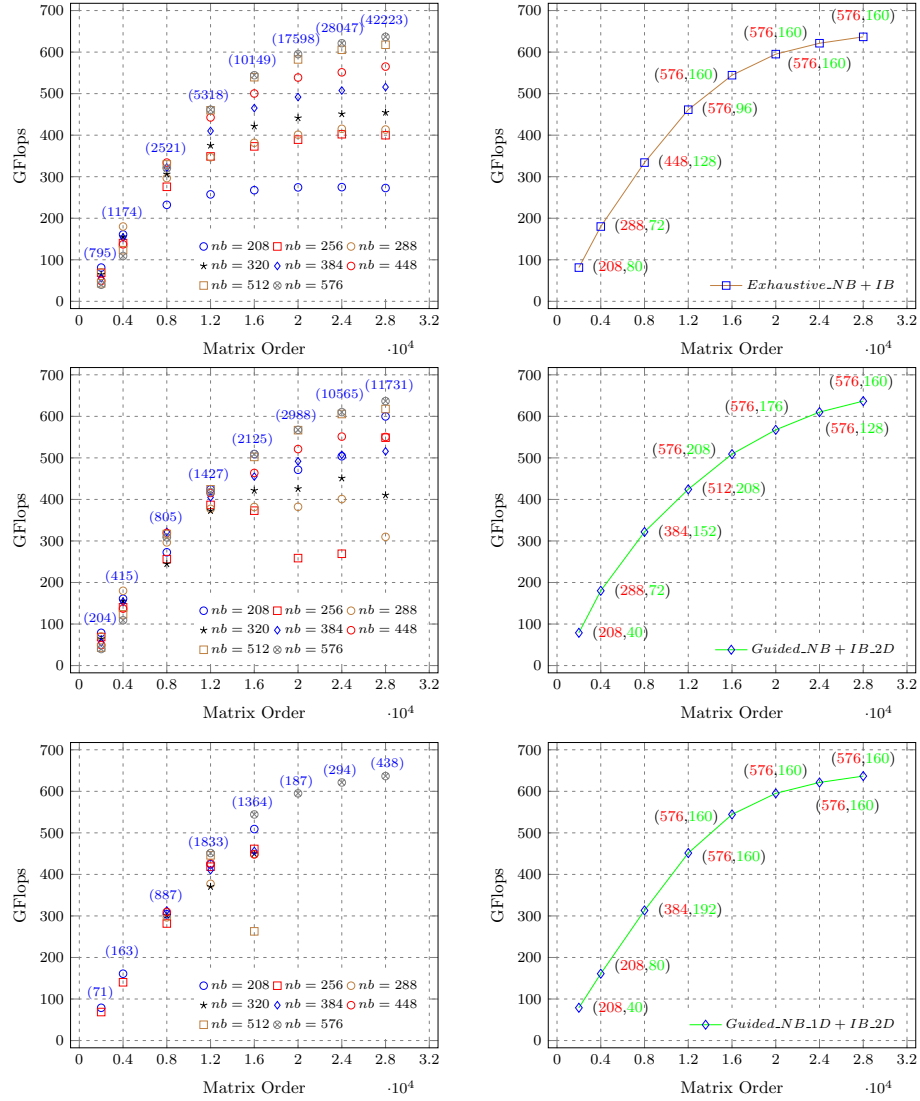
Fig. 5. QR factorization. Training time (left, in seconds) for each problem size when using the exhaustive (top left) and guided (middle and bottom) search strategies, and performance obtained (right) with each strategy when using the best (*nb*, *ib*) values for each problem size.
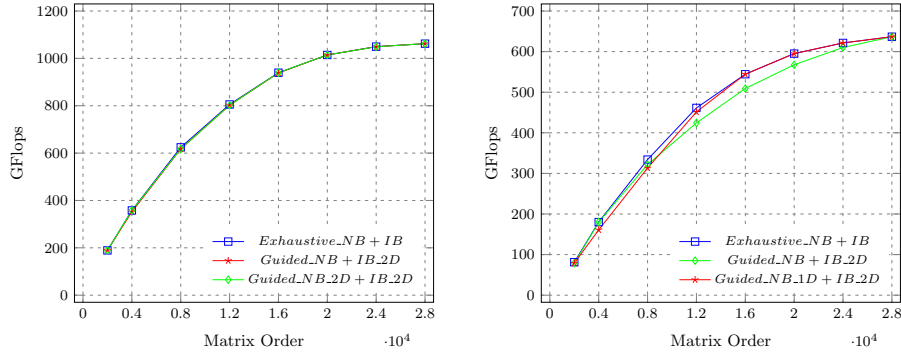
Fig. 6. Comparison of the performance obtained for the LU (left) and QR (right) routines of Chameleon with the selected (*nb*, *ib*) values for each problem size when using the exhaustive and guided search strategies.

Table 1. Performance obtained (in GFlops) with the Cholesky routine of Chameleon for each problem size using the best values for the block size and the system parameters (number of CPU cores and number of GPUs) together with the best scheduling policy.

n	nb	Cores	GPU-IDs	Scheduling Policy	Tuned Performance	Default Performance	Improvement (%)
1000	208	12	{}	eager	47	23	51
2000	208	9	{1,5,0}	lws	143	93	35
4000	288	7	{1,5,0,2,3}	ws	374	312	17
6000	320	6	{1,5,0,2,3,4}	dmdas	590	530	10
8000	320	6	{1,5,0,2,3,4}	dmdas	734	662	10

4.3   Validating the Methodology

Once the routines have been trained in the heterogeneous system and the best values for the *AP* have been stored in the database, the proposed methodology should be validated. As shown in Fig. 3, the validation process can be done in an empirical or simulated way. In both cases, the main goal is to show the performance obtained by the routine when the information stored in the database is used to select the values of the *AP* for a given problem size. The selection of the *AP* values is carried out based on the performance information stored for the closest problem size to the current one. Next, the routine is executed (or simulated) with the selected values.

Table 2 shows the results obtained with the QR routine of Chameleon using the empirical approach. A different set of problem sizes from the one used for the previously trained routine is considered to analyze how far the selected values for the *AP* and the performance obtained are from the experimental optima. It is noted that only two selected values slightly differ from the optimum ones, but its

impact on performance is negligible. Therefore, it proves that the methodology works as expected and could be applied to other linear algebra routines.

Table 2. Performance comparison for the QR routine of Chameleon when using the values for nb and ib selected by the methodology for each problem size vs. using the optimal ones after training the routine.

n	Autotuned			Optimal			Variation
	nb	ib	GFlops	nb	ib	GFlops	%
10000	512	112	399	512	112	399	0
14000	576	128	507	512	128	508	0
18000	576	160	573	576	160	573	0
22000	576	160	610	576	160	610	0
26000	576	160	626	576	192	626	0

5   Conclusions

This work presents a methodology to self-optimize task-based libraries, such as Chameleon. A set of strategies to search the best values for different adjustable parameters have been applied during the training phase to several linear algebra routines, such as the Cholesky, LU and QR factorizations.

On the one hand, the impact on performance of certain parameters of the routines, such as the block size and the inner block size, is analysed. The proposed search strategies allow to obtain good performance, reducing even the training time of the routines by guiding the search for the optimal values of these parameters. Also, additional adjustable parameters have been considered, such as the number of computing resources of the heterogeneous platform and the scheduling policy of the runtime system, showing the importance of properly selecting the value of each of them to further reduce the execution time of the routines. Our intention is to extend this work by using the SimGrid simulator in the training and validation phases of the proposed methodology to analyze the potential benefits of its use, as well as to integrate the hierarchical autotuning approach proposed in [9] within the Chameleon library.

References

1. Agullo, E., Augonnet, C., Dongarra, J., Ltaief, H., Namyst, R., Thibault, S.: Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In: GPU Computing Gems. vol. 2 (2010)

2. Agullo, E., Aumage, O., Faverge, M., Furmento, N., Pruvost, F., Sergent, M., Thibault, S.P.: Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model. IEEE Transactions on Parallel and Distributed Systems (2017)

3. Agullo, E., Cámara, J., Cuenca, J., Giménez, D.: On the Autotuning of Task-Based Numerical Libraries for Heterogeneous Architectures. In: Advances in Parallel Computing. vol. 36, pp. 157–166 (2020)

4. Anzt, H., Haugen, B., Kurzak, J., Luszczek, P., Dongarra, J.: Experiences in Autotuning Matrix Multiplication for Energy Minimization on GPUs. Concurrency and Computation: Practice and Experience **27**, 5096–5113 (2015)

5. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A.: STARPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. Concurrency and Computation: Practice and Experience **23**(2), 187–198 (2011)

6. Bilmes, J., Asanovic, K., Chin, C.W., Demmel, J.: Optimizing Matrix Multiply Using PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology. In: Proceedings of the 11th International Conference on Supercomputing. pp. 340–347 (1997)

7. Buttari, A., Langou, J., Kurzak, J., Dongarra, J.: A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. Parallel Computing **35**, 38–53 (2009)

8. Cámara, J., Cuenca, J., García, L.P., Giménez, D.: Empirical Modelling of Linear Algebra Shared-Memory Routines. Procedia Computer Science **18**, 110–119 (2013)

9. Cámara, J., Cuenca, J., Giménez, D.: Integrating software and hardware hierarchies in an autotuning method for parallel routines in heterogeneous clusters. The Journal of Supercomputing **76**(12), 9922–9941 (2020)

10. Intel MKL.: http://software.intel.com/en-us/intel-mkl/

11. Kelefouras, V., Kritikakou, A., Goutis, C.: A Matrix-Matrix Multiplication Methodology for Single/Multi-Core Architectures Using SIMD. The Journal of Supercomputing **68**(3), 1418—1440 (2014)

12. Low, T.M., Igual, F.D., Smith, T.M., Quintana-Orti, E.S.: Analytical Modeling Is Enough for High-Performance BLIS. ACM Transactions on Mathematical Software **43**(2) (2016)

13. NVIDIA cuBLAS.: https://docs.nvidia.com/cuda/cublas/index.html

14. Stanisic, L., Thibault, S., Legrand, A.: Faithful Performance Prediction of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures. Concurrency and Computation: Practice and Experience **27**, 4075–4090 (2015)

15. Tomov, S., Dongarra, J.: Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems. Parallel Computing **36**, 232–240 (2010)

16. Videau, B., Pouget, K., Genovese, L., Deutsch, T., Komatitsch, D., Desprez, F., Méhaut, J.F.: BOAST: A Metaprogramming Framework to Produce Portable and Efficient Computing Kernels for HPC Applications. The International Journal of High Performance Computing Applications **32**, 28–44 (2018)

17. Whaley, C., Petitet, A., Dongarra, J.: Automated empirical optimizations of software and the ATLAS project. Parallel Computing **27**, 3–35 (2001)