The Journal of Supercomputing (2020) 76:9922–9941
https://doi.org/10.1007/s11227-020-03235-9

Integrating software and hardware hierarchies in an autotuning method for parallel routines in heterogeneous clusters

Jesús Cámara[1] · Javier Cuenca[1] · Domingo Giménez[2]

Published online: 7 March 2020

Abstract
A hierarchical approach for autotuning linear algebra routines on heterogeneous platforms is presented. Hierarchy helps to alleviate the difficulties of tuning parallel routines for high-performance computing systems. This paper analyzes the application of the hierarchical approach at both the hardware and software levels, using the basic matrix multiplication and the Strassen multiplication as proof of concept on multicore+coprocessor nodes. In this way, the hierarchical approach allows partial delegation of the efficient exploitation of the computing units in the node to the underlying direct autotuned matrix multiplication used in the base case.

Keywords Autotuning · Hybrid programming · Heterogeneous computing · Multicore · Manycore

1 Introduction

Today, standard computational nodes include one multicore CPU together with one or more coprocessors (typically GPUs and/or Many Integrated Core, e.g., the Intel Xeon Phi). The basic computational components of these nodes have different architectures and computational capacities; therefore, they can be organized/managed hierarchically, with the basic computing units (CPU, GPU and MIC) having separate memory spaces and communicating with data transfers between them across

✉ Javier Cuenca
jcuenca@um.es

Jesús Cámara
jcamara@um.es

Domingo Giménez
domingo@um.es

[1] Department of Engineering and Technology of Computers, University of Murcia, Murcia, Spain

[2] Department of Computing and Systems, University of Murcia, Murcia, Spain

the memory associated with the CPUs and those of the coprocessors. This heterogeneous and hierarchical organization makes the efficient exploitation of routines for those nodes difficult and requires techniques for exploiting the underlying heterogeneity and hierarchy.

Elsewhere, linear algebra routines are widely used as basic computational kernels in scientific software, and their optimization for today's standard heterogeneous nodes would lead to important improvements when solving scientific problems based on highly efficient linear algebra libraries such as MKL [16], PLASMA [19], MAGMA [1] and Chameleon [8], whose routines base their optimization in implementations by blocks or tiles in which the basic kernel is a highly optimized matrix multiplication [13]. The matrix multiplication has been widely researched, and there are now many highly efficient implementations for today's systems [14, 15, 17]. As with computational systems, the optimization of linear algebra routines has traditionally been based on a hierarchical schema [6], with a set of basic linear algebra routines (BLAS) and higher-level routines (LAPACK) developed by blocks or tiles.

A hierarchical and decentralized schema can be applied for the automatic optimization of linear algebra software for heterogeneous CPU+multicoprocessor nodes [7]. This paper analyzes the application of the hierarchical approach to both the hardware and the software in multicore+multicoprocessor nodes and uses a traditional three-loop matrix multiplication and a Strassen multiplication as proof of concept.

The rest of the paper is organized as follows. Section 2 makes a brief comparison with other hierarchical optimization approaches. The implementations of the parallel routines used to illustrate the methodology of hierarchical autotuning for linear algebra (the traditional and the Strassen matrix multiplications) are presented in Sect. 3. Section 4 outlines the general ideas of the hierarchical methodology, and Sect. 5 gives some experimental results which show the applicability of the proposal. Section 6 comments on some possible extensions of the methodology, and Sect. 7 concludes the paper.

2 Related work

Traditionally, hierarchical approaches have been applied in the design of general software [4], and, particularly, in the design of parallel linear algebra routines [6] and also in the theoretical study of its execution time [11]. Recently, they have been applied in the development of linear algebra routines for both clusters and distributed systems. The autotuning process applied to these routines must take into account this degree of heterogeneity when searching for the best adjustable parameter values [23].

The search for satisfactory values for those adjustable parameters when tuning a routine for a hierarchical computational system can be improved in various ways. The convex form of the function that represents the execution time [24] or accurate theoretical models [5] can be used. An optimization problem of finding the values of the parameters which minimize the execution time can be approached through

metaheuristic methods (e.g., OpenTuner [2]) or with heuristics tailored for each specific routine to optimize [10].

Some works exploit heterogeneity to improve this search. When applying OpenTuner, the parameters can be grouped hierarchically and the tool can be applied incrementally to the parameters at different levels. In [15], the hierarchical organization of the communication scheme is exploited for the optimization of a parallel matrix multiplication.

In [18], a study of each routine is carried out to perform a division of the search space of the adjustable parameters, a graph of dependencies is generated, and then, independent subspaces composed of the nodes of the graph may be tuned with individual search algorithms. In [22], the autotuning process is divided into different phases that work with different sets of adjustable parameters: local parameters such as loop unrolling, those related to the threads and, finally, those referring to the distribution of work between nodes.

As far as we know, our proposal is the first to consider a hierarchy in both the hardware and software, in a two-dimensional way. Furthermore, the set of parameters considered at each level form a black box that is autotuned separately and can be reused in the autotuning process at any upper level box, which includes it in the hardware or software hierarchy. In addition, variability is an important issue in autotuning [20], and the hierarchical optimization methodology used here allows us to intensify the experimentation at different levels depending on the degree of variability observed for each routine and for each computing unit, and different optimization methods (theoretical, experimental, hybrid) or software (e.g., OpenTuner) can be used at different levels depending on how accurate and efficient they prove to be for the particular software and hardware.

3 Parallel implementations of the naive three-loop multiplication and the Strassen method

The basic and the Strassen matrix multiplication are used to show the application of the hierarchical autotuning methodology. The development of routines which are competitive with efficient implementations of the matrix multiplication is out of the scope of this paper.

3.1 Matrix multiplication

Our implementation of the basic matrix multiplication for multicore+coprocessor is heterogeneous. The data are unevenly distributed between the computing units (CUs) in the node, and the computations (matrix multiplication) are carried out in each CU with optimized routines. The amount of work to assign to each CU is obtained taking in consideration their relative speed.

For the matrix multiplication $C = \alpha AB + \beta C$, $A \in R^{m \times k}$, $B \in R^{k \times n}$, $C \in R^{m \times n}$, matrix A is replicated in all the CUs in the node, and matrix B is scattered in adjacent blocks of columns. There are some works on techniques for balancing

the workload between the CPU and the coprocessors [12, 24]. If the node comprises one multicore CPU and c coprocessors (referenced from 1 to c), the matrix multiplication can be expressed as $C = \alpha(AB_1 | \cdots | AB_{c+1}) + \beta(C_1 | \cdots | C_{c+1})$, where $\alpha AB_i + \beta C_i$, with $1 \leq j \leq c$, is assigned to the coprocessor j, and $\alpha AB_{c+1} + \beta C_{c+1}$ to the CPU, with $B_i \in R^{k \times n_i}$, $\sum_{i=1}^{c+1} n_i = n$.

In a node with one multicore CPU and c coprocessors, of which g are GPU and m MIC ($c = g + m$), our task is to determine the optimum values of n_i, $1 \leq i \leq c + 1$. Thus, it depends on the speed of the CUs in the node when running the basic matrix multiplication when optimized for the current matrix sizes. We use the basic multiplication from MKL [16] for CPU and Xeon Phi and the one from cuBLAS [9] for GPU, but the methodology can be applied with other basic libraries and for other versions of the matrix multiplication.

When the matrix multiplication is executed on a platform composed of N heterogeneous nodes, where node i has a multicore CPU (with p_i cores) and c_i accelerators, the set of adjustable parameters, whose values must be chosen for setting the execution of the routine, is:

$$\begin{aligned}\{&n_{1,1}, \ldots, n_{1,j}, \ldots, n_{1,c_1}, n_{1,\text{cpu}}, \ldots, n_{i,1}, \ldots, n_{i,j}, \ldots, n_{i,c_i}, n_{i,\text{cpu}}, \ldots,\\ &n_{N,1}, \ldots, n_{N,j}, \ldots, n_{N,c_N}, n_{N,\text{cpu}}\},\end{aligned} \tag{1}$$

with $n_{i,j}$ being the number of columns of matrix B mapped to the accelerator j of node i ($1 \leq i \leq N$, $1 \leq j \leq c_i$) and $n_{i,\text{cpu}}$ the part mapped to the CPU in node i. The search space for the best values for these adjustable parameters following a global method would be huge. For example, if matrix B is divided into blocks with s consecutive columns, the number of possible assignations has an order of $O\left(\left(N + \sum_{i=1}^{N} c_i\right)^{\frac{n}{s}}\right)$. Alternatively, with a hierarchical approach, the search space to decide the distribution between nodes has an order of $O\left(N^{\frac{n}{s}}\right)$, and in each node i the order is $O\left(\left(1 + c_i\right)^{\frac{n}{s}}\right)$. If a heuristics method based on the exploration of the neighborhood is used to guide the search [10] and the neighbors for a given configuration are those obtained just by transferring a quantity of workload from one element to other, the neighborhood has an order of $O\left(\left(N + \sum_{i=1}^{N} c_i\right)^2\right)$ without the hierarchical approach, and $O\left(N^2\right)$ and $O\left(\left(1 + c_i\right)^2\right)$ for the whole platform and the nodes with the hierarchical methodology.

Besides, if we take into account the set of adjustable parameters for each basic computational element, the search space grows considerably. For instance, when using the MKL multiplication, the number of MKL threads influences the execution time; therefore, the value for this parameter must be considered for CPU and Xeon Phi. So, if $t_{i,j}$ represents the number of threads to use in each element j of each node i, the set of parameters in Eq. (1) is now

$$\begin{aligned}\{&n_{1,1},\ldots,n_{1,j},\ldots,n_{1,c_1},n_{1,\text{cpu}},\ldots,n_{i,1},\ldots,n_{i,j},\ldots,n_{i,c_i},n_{i,\text{cpu}},\ldots,\\&n_{N,1},\ldots,n_{N,j},\ldots,n_{N,c_N},n_{N,\text{cpu}},\\&t_{1,1},\ldots,t_{1,j},\ldots,t_{1,c_1},t_{1,\text{cpu}},\ldots,t_{i,1},\ldots,t_{i,j},\ldots,t_{i,c_i},t_{i,\text{cpu}},\ldots,\\&t_{N,1},\ldots,t_{N,j},\ldots,t_{N,c_N},t_{N,\text{cpu}}\}\end{aligned} \tag{2}$$

3.2 Strassen multiplication

The Strassen multiplication has a recursive schema in which the basic operations are at the same time matrix multiplications, which are carried out with efficient implementations for the system in hand. It follows the typical divide-and-conquer recursive paradigm. We consider square matrices, and the matrices to be multiplied and the resulting matrix are divided into four submatrices of size $\frac{n}{2} \times \frac{n}{2}$. The routine is then called recursively seven times with matrices of this size. Ten additions and subtractions of matrices $\frac{n}{2} \times \frac{n}{2}$ are carried out to form the matrices to be multiplied, and the seven resulting matrices are combined with eight operations of the same type and size. For submatrices smaller than a base size, the direct heterogeneous multiplication is used.

When the Strassen multiplication is installed directly on a node, the parameters to be selected are those corresponding to the Strassen method (recursion level and assignation of the basic multiplications to the CUs of the node), and for each basic multiplication the parameters considered depending on the unit to which it is assigned (workload for each basic CU, and the number of threads on CPU and Xeon Phi, and no parameter for GPU).

4 Methodology for hierarchical autotuning

An autotuning methodology is used to provide linear algebra routines with automatic optimization capability for heterogeneous nodes. Thus, when the user wants to run the routine for a specific problem size, the execution is close to optimum in execution time without user intervention. Some decisions which depend on the computing units and the routine implementation need to be taken. They correspond to parameters (*Algorithmic Parameters*, AP) which determine the way in which the routine is run. The values of the parameters which give the lowest execution time are searched for, either theoretically or empirically. In complex computational systems, with a large number of CUs, the number of parameters and their possible values can be huge, making it impossible to explore exhaustively all the possible values for all the parameters (Eq. 2). The hierarchical approach helps to overcome this problem, organizing the computational units and/or the routines hierarchically, with smaller groups of decisions to take at each level of the hierarchy, and taking these decisions at each level using the information from previous levels.

4.1 Basis of the autotuning methodology

The hardware dimension is considered here in order to simplify the description of the autotuning methodology. The following subsection shows how this proposal is applied to a two-dimensional hardware+software hierarchy.

In general, a platform is considered as a computing unit, CU, of the highest level made up of a set of computing units of a lower level. Recursively, this approach is used for each CU at the successive levels until the basic level is reached. In a node, the whole node is considered at the highest level (level 1), and the basic computing units (CPU, GPU, MIC or any other accelerator) composing the node are considered at the lowest level (level 0). Additionally, the computing units of level $l-1$ which compose a unit at a level l are connected through links of level $l-1$, which correspond to how the units at level $l-1$ communicate data. Those links can be physical or logical. For example, the CUs can share the memory and the data can be communicated through the memory; or they can be nodes in a cluster with an interconnection network. In our case, the three types of computing units in a node communicate with transfers to and from the CPU and the coprocessors.

Given a routine, at each level l of the hierarchy, the search process for the best AP values is performed simultaneously in all the CUs of this level, for each problem size previously selected and stored in *Level_l_Installation_Set* by the platform manager. When the installation finishes for level l, the best AP values and the performance information obtained for each CU and for each link (bandwidth and latency) are stored in *Level_l_Performance_Information* (Fig. 1). This information is used to guide the process at level $l+1$, so reducing the complexity and the time of the installation process. As mentioned, the search for the best values of the AP for each CU can be made independently, which makes it easier to use different methods for searching, for each level and even for each CU. Some possible methods are:

- *TPM: Theoretical-Practical Modeling* In this approach, the working time, T_w, to solve the problem of size n_i in CU_i^l is considered to be the maximum execution time needed by its CUs of level $l-1$ for solving the portion of the problem assigned to each of them, $n_{i,j}$ according to the *Level_l-1_Performance_Information*. The execution time for each CU of level l, $CU_{i,j}^{l-1}$, is the addition of the communication time (T_c) for receiving operands and sending results back from/to the data source, $CU_{i,0}^{l-1}$, plus the working time for the solution of that subproblem, T_w. So, the theoretical optimum time is represented as

$$T_w(CU_i^l, n_i) = \min_{AP_i^l} \left\{ \max_{CU_{i,j}^{l-1}} \left\{ T_w(CU_{i,j}^{l-1}, n_{i,j}) + T_c(CU_{i,0}^{l-1}, n_{i,j}) \right\} \right\} \tag{3}$$

  With this approach, the installation process for the level l, with $l > 0$, does not entail additional experimentation, since it is based completely on the information from the previous level, *Level_l-1_Performance_Information*. The only experimentation necessary is that of level 0, to obtain the performance of the CUs and the time of the links at that basic level.

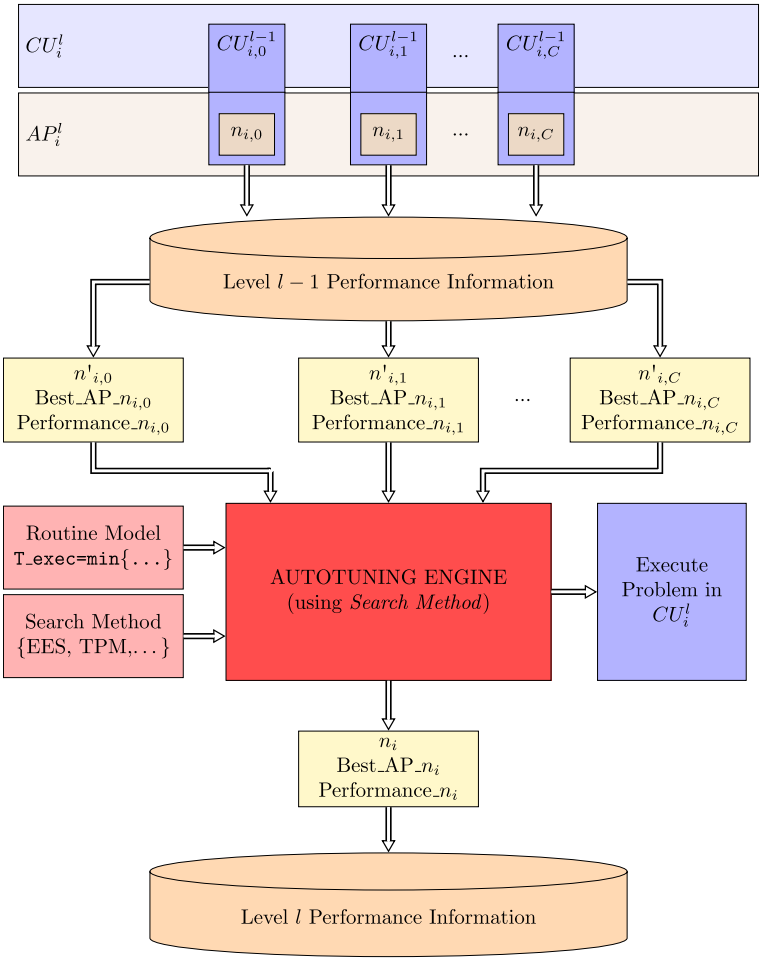

Fig. 1 Operation diagram of the autotuning engine for the CU i of level l, CU_i^l, searching the best values for its adjustable parameters, AP_i^l, for each problem size, n_i, at *Level_l_Installation_Set*

- *EES: Exhaustive Experimental Searching* The installation can be executed for the routine in CU_i^l for each subproblem of the sizes collected in *Level_l_Installation_Set* for this CU. For each of these sizes, the goal is to find the combination of values of the AP (in a range preset by the system manager) of that CU, AP_i^l, which provides the lowest execution time.
- *HES: Heuristic Experimental Searching* An exhaustive experimental search may be excessively costly, so it can be replaced by another type of experimental search that includes some kind of heuristics, for example, starting from the selection of the AP_i^l values made by the theoretical–practical modeling and performing a local experimental search around these values.

As described, the information of the CU of the previous level corresponds to the black boxes and, therefore, not have to be modified, but is used in the corresponding method chosen for CU_i^l. So, the search space is considerably reduced. In the same way, the autotuning work for each CU is isolated and modulated and can be done simultaneously for all the units at the same level, using the appropriate method for each of them.

4.2 Methodology for a two-dimensional hardware+software hierarchy

Like the hardware, the software is also organized in levels, with the basic routines at the lowest level, and the routines calling to those basic routines at a higher level. Any number of levels can be considered.

We illustrate the methodology with two levels for the hardware (a node as level 1 and its CUs as level 0) and two levels for the software (the basic matrix multiplication as level 0 routine and, as an example of level 1 routine, the Strassen multiplication).

There are different possibilities when installing a routine of a certain level on a computing platform of a particular level. Figure 2 shows the possibilities for various levels of routines and platforms. A circle in the figure (labeled `SRHP`) represents that the routine `S` of level `R` is installed in platform `H` of level `P`. An arrow labeled `SRHP-srhp` represents that routine `S` of level `R` is installed in platform `H` of level `P` using the information obtained when the routine `s` of level `r` was installed in platform `h` of level `p` (*Level_srhp_Peformance_Information*). Only two levels (continuous lines) are explained in detail, and some possible extensions (dashed lines) are commented on elsewhere. A bottom-up approach is used to show how the methodology works starting from the lowest level (`S0H0`) to the highest level (`S1H1`):

- Level `S0H0`: The installation of the basic routine in each basic computing unit is carried out by searching for the best AP values for the problem sizes previously

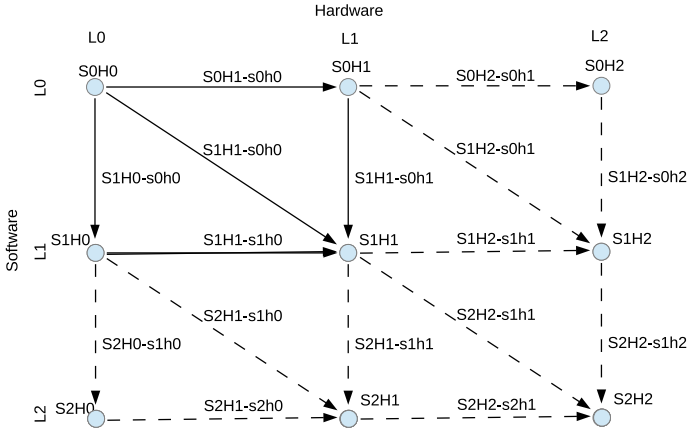

Fig. 2 Possibilities of installation of routines for various levels of software and hardware

stored in *Level_S0H0_Installation_Set*. Since `S0H0` is the lowest level of the hierarchy, the search method employed for this level is exhaustive. The values of the AP for which the lowest experimental time is obtained are stored together with the performance (GFlops) in *Level_S0H0_Performance_Information*. In our example, no parameters are considered for GPU, and only the performance is stored. For CPU and MIC, the number of threads is also selected.

- Level `S0H1`: When a routine of level 0, `S0`, is installed on the node, `H1`, the amount of work to assign to each CU of the node needs to be determined, as well as the parameters for each particular CU for the problem size assigned to it. The values of these parameters could be selected either by working directly at level 1 (`S0H1`) or by the hierarchical approach delegating the selection of the parameters at this level to the installation of the previous level (`S0H1-s0h0`). In order to select the best theoretical partition of the work, the hierarchical approach can be applied to obtain the information from the previous level (*Level_S0H0_Performance_Information*) and then perform experiments varying the amount of work assigned to each CU (EES or HES method) or use Eq. (3) without experimentation (TPM method).
- Level `S1H0`: The installation of the level 1 routine, `S1` (Strassen multiplication in our example), in a CU of level 0 could be made at the highest level with experiments, varying the AP values for both this routine and the basic routine. With the hierarchical approach (`S1H0-s0h0`), only the parameter value of the level 1 routine (Strassen recursion level) is searched for, whereas the parameter values of the basic routine (number of threads for CPU or MIC) would be taken from the information generated by `S0H0` (*Level_S0H0_Performance_Information*). In this way, the performance information from a lower level (`S0H0`) is reused for higher levels of hardware (`S0H1-s0h0`) or software (`S1H0-s0h0`) and could also be reused for other nodes with the same basic CUs or for other routines which call to the basic matrix multiplication with sizes and shapes similar to those used in the installation in `S0H0` (*Level_S0H0_Installation_Set*).
- Level `S1H1`: The installation at the highest level can be made hierarchically in three ways:
  - `S1H1-s0h1`: The routine `S1` uses `S0` optimized for the level 1 of the hardware, so the exploitation of the heterogeneity of the system is delegated to `S0`. Only AP values of `S1` have to be selected. In our example, the basic matrix multiplication optimized for the node is used to perform each multiplication in the base case of the Strassen recursion. The distribution of the data to the CUs is made inside the basic multiplication. Therefore, the only parameter value to be determined is the recursion level of the Strassen method.
  - `S1H1-s0h0`: Each basic routine, `S0`, used in `S1` is assigned to a single CU of level 0. The decision on how to perform this mapping is a parameter whose value must be determined. An exhaustive experimental search (EES method) for the best allocation among all the possible ones may not be viable. An alternative is to perform a theoretical search based on the performance information of each basic CU obtained and saved when `S0` has been installed at level `H0` (TPM method). In the example, each basic multiplication is per-

formed in a single CU, with the parameters (number of threads in CPU and MIC) stored in `S0H0` for the matrix sizes closest to those of the matrices to be multiplied. So, the preferred recursion level is selected together with the distribution of the basic multiplications between the basic CUs in the node. Another alternative is to use dynamic scheduling [3]. The basic matrices are distributed among the CUs dynamically, starting with the fastest units, and the information from `S0H0` can be used to decide the CU in which the basic multiplications are assigned. The working distribution with lowest computing inactivity gaps is searched for [21].

- `S1H1-s1h0`: This case is similar to `S1H1-s0h0`, with the only difference that now, `S0` (the basic multiplication) is replaced by `S1` (a Strassen multiplication), for which different values of its AP (recursion level) may have been selected for each computing unit when `S1` was installed on it with `S1H0`.

The three possibilities can be tested (experimentally or theoretically) to select the best of the three configurations: successive basic multiplications which exploit the heterogeneity in the node, and sets of basic or Strassen multiplications assigned to the basic CUs. Each possibility corresponds to a different implementation of the Strassen method for the node. Furthermore, the schema in Fig. 2 follows the dynamic programming paradigm, with the value for levels 1 for software and hardware obtained from the optima stored for lower levels.

Table 1 summarizes the AP for levels 0 and 1 of hardware (multicore, GPU and MIC as CUs at level 0, and nodes composed of level 0 CUs at level 1) and software (basic multiplication at level 0, Strassen multiplication at level 1).

The parameters for the basic matrix multiplication are those in Eq. (2). For the Strassen multiplication, the APs are the recursion level and the parallel implementation of Strassen to be used (`s0h1`, `s0h0` or `s1h0`). For the two Strassen

Table 1 Algorithmic parameters, AP, for levels 0 and 1 in the hierarchy (Fig. 2), with the basic matrix multiplication as level 0 routine and the Strassen multiplication as an example of level 1 routine

S0H0	*S0H1*
No. of threads in multicore or MIC ($t_{i,j}$ in Eq. 2)	Work distribution between CUs of `H0` ($n_{i,j}$ in Eq. 2)
S1H0	*S1H1*
Strassen recursion level	Hierarchical implementation scheme: (a) `S1H1-s0h1` Strassen recursion level (b) `S1H1-s0h0` Strassen recursion level Mapping: basic multiplications to CUs of `H0` (c) `S1H1-s1h0` Mapping: Strassen multiplications to CUs of `H0`

implementations based on simultaneous execution of routines optimized for `H0` (`s0h0` or `s1h0`), an additional parameter would be the mapping of these routines to the CUs of level 0.

5 Experimental results

This section shows some results to illustrate the installation of some linear algebra routines using a hierarchical autotuning engine at different levels, as shown in Fig. 2. The basic matrix multiplication and the Strassen multiplication are used for software levels 0 and 1, and level 1 of hardware corresponds to a hybrid node with a multicore CPU and several coprocessors such as GPUs and/or Xeon Phi (placed at level 0). Experiments have been carried out in four computing nodes:

- ***6c_GTX480***: 1 CPU AMD Phenom II X6 1075T (6 cores) and 1 GPU NVIDIA GForce GTX 480 Fermi.
- ***24c_K20c***: 4 CPU Intel Xeon E7530 (hexa-core) (24 cores) and 1 GPU NVIDIA Tesla K20c (Kepler).
- ***12c_2C2075_4GTX590***: 2 CPU Intel Xeon E5-2620 (hexa-core) (12 cores), 2 GPUs NVIDIA Tesla C2075 Fermi and 4 GPUs NVIDIA GeForce GTX 590 Fermi.
- ***12c_GT640K_2MIC***: 2 CPU Intel Xeon E5-2620 (hexa-core) (12 cores), 1 GPU NVIDIA GeForce GT 640 Kepler and 2 Intel Xeon Phi 3120A KNC.

MKL was used for the basic multiplications on CPU and Xeon Phi, and cuBLAS was used for GPU, but the methodology works in the same way for other basic libraries, and the results are similar.

How the hierarchical autotuning works is shown. At the lowest level (`S0H0`), experiments are carried out for each problem size in the *Installation_Set* and each basic CU considered. The sizes in the *Installation_Set* must be representative of the sizes with which the matrix multiplication will be called at higher levels. Hence, square and rectangular matrices need to be experimented with. For example, for multiplications of size 1000, in the multiplication on a node, matrix B may be partitioned in different sizes, and multiplications of sizes $1000 \times 1000 \times n$, for several values of $n \leq 1000$, are included in the *Installation_Set*. When more values of n are used, the installation time increases, but the information generated represents the behavior of the routine better. Table 2 shows the installation time for the different

Table 2 Installation time (in s) on each CU, for an *Installation_Set* with sizes s, s, n, with $s = 1000, 2000, 3000, 4000, 5000, 6000$ and $n = k\frac{s}{50}$, $k = 1, 2, \ldots, 50$

6c_GTX480		***24c_K20c***		***12c_2C2075_4GTX590***			***12c_GT640K_2MIC***		
CPU	GTX480	CPU	K20	CPU	C2075	GTX590	CPU	MIC	GT640
8491	791	29147	133	9655	183	297	2867	9707	1241

CUs of each computing node considered for sizes 1000, 2000, 3000, 4000, 5000, 6000 and 50 intermediate values of n for each size s ($\frac{s}{50}, 2\frac{s}{50}, \ldots, s$). The total installation time is the highest of the times in all the CUs (the installation is made in parallel). The time is around 9 h, which is affordable because it runs just once, but a HES method with guided search can be used to reduce this time [10].

On advancing one level in the hardware, for an installation `S0H1-s0h0` the information of lower level can be used at least two ways:

- At this level, we consider an *Installation_Set* with square matrices of sizes 1500, 2500, 3500, 4500, 5500, 6500 and 7500. The workload to assign to each CU is decided for each matrix size with the relative performance of each unit stored for the closest problem size in the *Installation_Set* of `S0H0` (for example, for 1500 the values associated with 1000 are considered). Figure 3 shows,

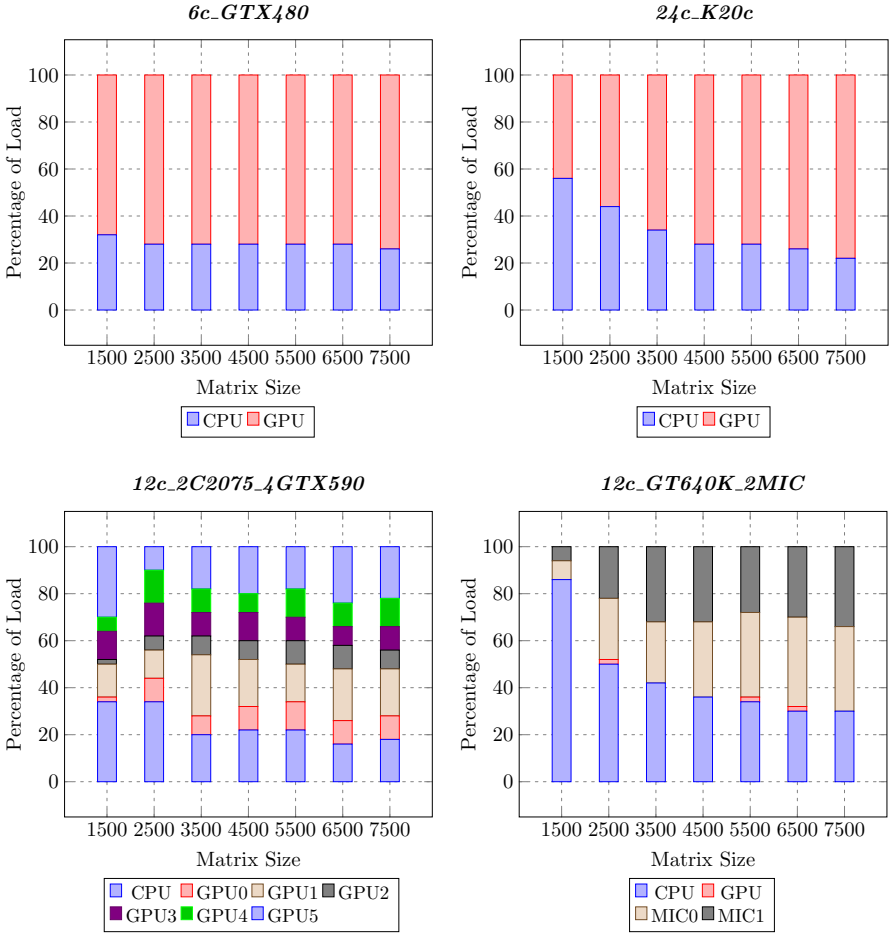

Fig. 3 Workload distribution among the CUs in the four nodes considered for different matrix sizes

for the four nodes considered, the workload distribution to each CU. The workload changes for the different nodes. For large problems, it tends to be proportional to the relative computational capacity of the units, but this is not so for smaller problems (especially in ***24c_K20c*** and ***12c_GT640K_2MIC***), which can be closer to the sizes used when the routine is used inside routines of higher level. In ***12c_2C2075_4GTX590***, the fastest GPUs are GPU1 and GPU5, and the GPU of ***12c_GT640K_2MIC*** is much slower than the other units, so the amount of work assigned to it is residual. No experiments are carried out in the installation, for which the information from `S0H0` is used, so the installation time is very low if the node comprises only a few CUs. For more CUs, this time increases. In ***12c_GT640K_2MIC***, with four units, it is around 0.28 s, and in ***12c_2C2075_4GTX590*** with seven CUs, it grows to 126.35 s.

- The prediction based only on the performance of the CUs without considering transfers (TPM method in `S0H0` performing experiments only on the computation) gives estimations far from the experimental measures. An installation with experiments which include transfers produces more accurate predictions and consequently facilitates better decisions. One possibility to reuse information from level 0 is to experimentally measure the transfer cost between CPU and coprocessors and to use accurate models of the execution time including the transfers cost [5]. The time is the maximum of the computational plus transfer times from all the CUs according to Eq. (3) (TPM method in `S0H0` performing experiments on computation and communications). Figure 4 compares in ***12c_2C2075_4GTX590*** and ***12c_GT640K_ 2MIC*** the prediction with and without transfers. The installation is made at level 0 and 1 with sizes 1000, 2000, 3000, 4000, 5000 and 6000 (and the corresponding rectangular matrices), and square multiplications of sizes 1500, 3500 and 5500 are used for the comparison. Without transfers, the number of columns of matrix B assigned to each CU is obtained with Eq. (3) with $T_c = 0$, and when the transfers are considered, the GFlops for each entry of the *Installation_Set* are obtained with executions for

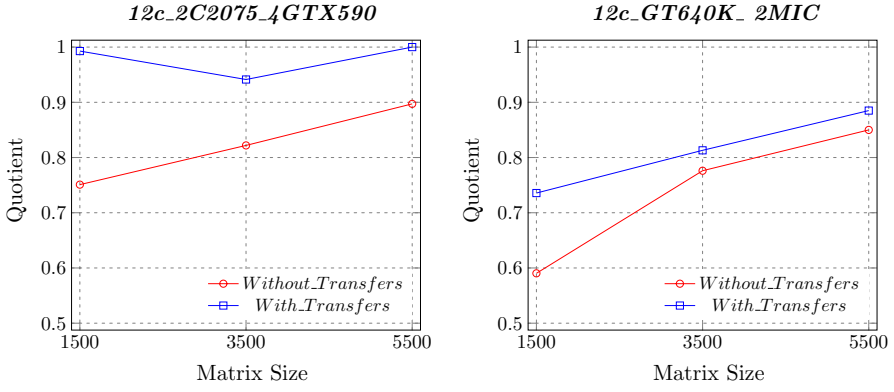

Fig. 4 Quotient of the execution time with respect to the lowest experimental, with the workload distribution selected depending on whether the transfers are taken into account or not in the execution time model (Eq. 3)

each size and workload distribution in the equation, so the values T_c are considered. This figure shows, for each test size, the quotient of the execution time with respect to the lowest experimental, which is obtained with exhaustive executions and varying the workload distribution among CUs. The decisions are better (quotient closer to 1) when the transfers are considered, and improve when the problem size increases.

When the level of the software increases (`S1H0`), we consider the Strassen multiplication in individual CUs. The optimum recursion level is obtained experimentally for the sizes in the *Installation_Set* on each CPU, GPU and Xeon Phi, and the parameters of the basic routine are delegated to the installation in the previous level of software (`S1H0-s0h0`). Figure 5 shows the quotient of the execution time of the direct multiplication with respect to that of the Strassen algorithm, when varying the recursion level and the number of threads, for matrix sizes 8000 and 12,000, for

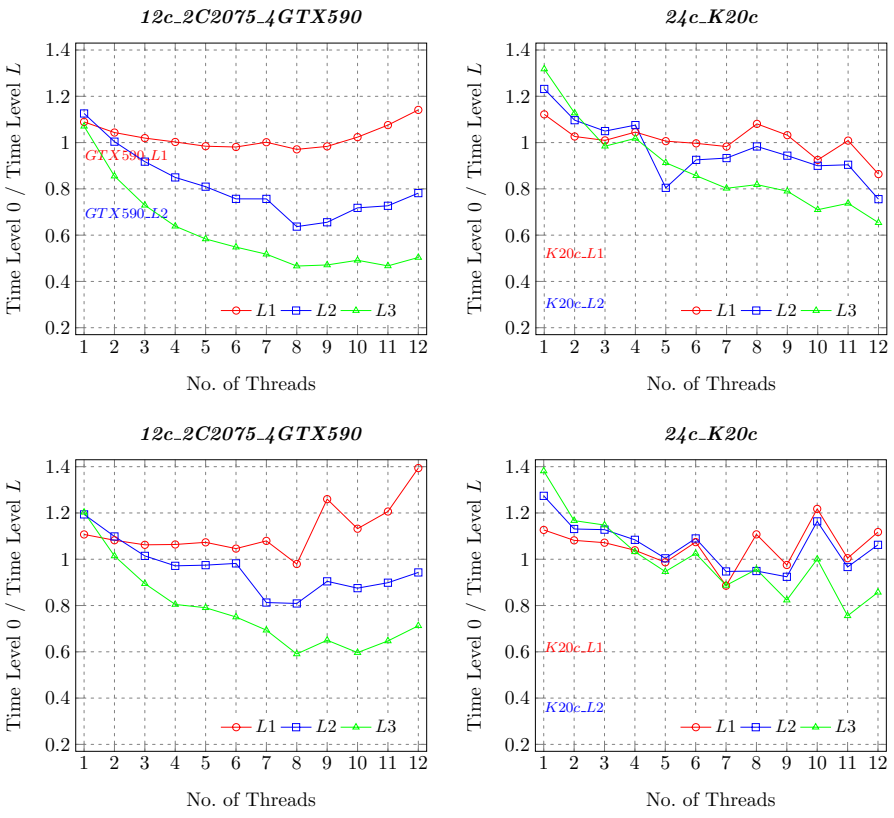

Fig. 5 Quotient of the execution time of the direct multiplication with respect to that of the Strassen's algorithm with recursion levels $L1$, $L2$ and $L3$, for matrix sizes 8000 (up) and 12,000 (bottom), in two nodes, with one GPU on each node

the nodes ***12c_2C2075_4GTX590*** and ***K20c***. The quotient for the slowest (***GTX590***) and the fastest (***K20c***) GPUs are also shown (for ***GTX590*** only size 8000 due to memory constraints). According to the figure, due to the higher speed of the GPUs, the selected level would be 0 (direct multiplication), whereas for the CPUs the preferred level for large matrix sizes is 1.

At the highest level of hardware and software, the Strassen multiplication can run with successive executions of the basic multiplication using the whole node in the basic case, so the only parameter to be selected is the level of recursion, since the workload distribution to the CUs is done in the basic multiplication (`S1H1-s0h1`). Table 3 shows, for different matrix sizes, the configuration with which the lowest execution time is obtained as well as the parameters selected. The results are for ***12c_2C2075_4GTX590***, and the use of all the GPUs in the node generates many transfers and a degradation on the performance, so different combinations of CPU and GPUs are considered (CPU+C2075, CPU+GTX590, C2075+GTX590, CPU+C2075+GTX590 and CPU+2C2075). The lowest times are always obtained with CPU+2C2075, which is the configuration with the highest computational capacity. For small matrices, the direct method is preferred, but due to memory limitations, Strassen with one recursion level can be used for larger matrices. For very large matrices, two levels are required. The workload of the Strassen method corresponds to that of the basic multiplication: half the size of the matrix or a quarter, for levels 1 and 2. The parameters are autonomously selected by the heterogeneous multiplication on the node (`S0H1`).

6 Possible extensions

The methodology can be extended in both hardware and software hierarchies (dashed lines in Fig. 2).

One possibility of extension in the hardware hierarchy is to use of the concept of subnodes (subsets of CUs) inside a node. In this way, each subnode becomes a CU of level 1 and the node is at level 2. The handling of the subnode concept introduces

Table 3 Method with which the lowest execution time is obtained, varying the matrix size on ***12c_2C2075_4GTX590***

Size	Best method	Parameter selection for basic multiplications				
		Combination of CUs	CPU threads	Workload distribution		
				CPU	1st C2075	2nd C2075
2000	Direct	CPU + 2 C2075	7	240	880	880
4000	Direct	CPU + 2 C2075	12	880	1520	1600
6000	StrassenL1	CPU + 2 C2075	11	600	1200	1200
8000	StrassenL1	CPU + 2 C2075	12	880	1520	1600
10,000	StrassenL1	CPU + 2 C2075	10	1000	2000	2000
12,000	StrassenL2	CPU + 2 C2075	11	600	1200	1200

The parameter selection in the basic multiplication is shown

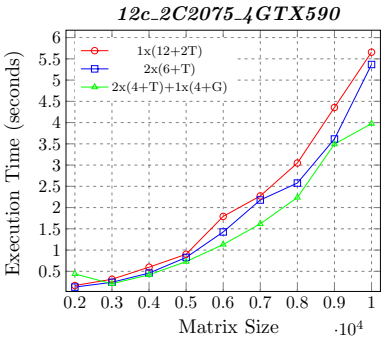

Fig. 6 Execution time with different hierarchical configurations of the computational node as a group of subnodes, varying the matrix size in ***12c_2C2075_4GTX590*** with two C2075 and one GTX590

Table 4 Workload distribution of the matrix multiplication in a cluster with four nodes

Size	***6c_GTX480***	***24c_K20c***	***12c_2C2075_4GTX590***	***12c_GT640K_2MIC***
1750	350	350	700	350
2750	550	825	1375	0
3750	750	1125	1500	375
4750	950	1425	1900	475
5750	575	1725	2300	1150

more flexibility to the proposal. When several routines without temporal dependencies are executed, there are two options: to execute one after the other using the node as a whole for each one of them, or to manage the node as a set of subnodes where simultaneous executions of these routines are mapped.

In the example, we have a `S1H2` strategy, in which the Strassen multiplication is optimized for the node with dynamic assignation of the basic multiplications to CUs of level 1 (subnodes). Figure 6 compares the execution time of the Strassen multiplication of level 1 on ***12c_2C2075_4GTX590*** for different configurations of subnodes: $1 \times (12c + 2C2075)$, one subnode composed of 12 CPU cores plus 2 C2075; $2 \times (6c + C2075)$, two subnodes, each composed of 6 CPU cores plus 1 C2075 card; and $2 \times (4c + C2075) + 1 \times (4c + GTX590)$, three subnodes, each with 4 CPU cores, two of them with one C2075 and the other with one GTX590. The extension of the hierarchy in the hardware contributes to lower execution times. Furthermore, the workload distribution to the basic CUs of each subnode is decided with information from installation of the basic multiplication on each subnode. For example, for matrix size 10,000, the executions with C2075 distribute 1000 columns of matrix B to the CPU and 4000 columns to the GPU, and in GTX590 the distribution is 1600 for CPU and 3400 to the GPU.

Another possibility of extension in the hardware hierarchy is to treat the whole cluster of nodes as a CU of level 2. A basic matrix multiplication which distributes matrix B among the nodes is a level 0 routine running on a level 2 CU (`S0H2`). Table 4 shows the workload distribution in a cluster made up of four nodes connected through Gigabit Ethernet. The installation time is short (less than 1 min) thanks to the use of the

information stored at level 1. The deviation of the performance with respect to the lowest experimental (exhaustive experiments varying the workload) is maintained at an acceptable level (between 2 and 10%) without user intervention.

Regarding extensions in the software hierarchy, other linear algebra routines can be optimized at level 1 using the same information of the installation for lower level routines. As an example, we consider an LU factorization by blocks on ***12c_2C2075_4GTX590*** (`S1H1`) which uses the matrix multiplication on each basic CU (`S0H0`). Table 5 shows the workload distribution and the GFlops achieved with the four multiplications inside the LU factorization of size $10{,}000 \times 10{,}000$ with a block size of 1000. The division shows the number of columns of matrix B distributed to the CPU and to each GPU. The six GPUs in the node are considered, but, due to the high CPU–GPU transfer cost, only three are used. The first and third GPUs are GTX590, and the second is a C2075, so more work is assigned to the C2075. When the size of the multiplication decreases, the same happens with the performance and the number of GPUs selected to work with.

Another aspect of possible extensions to the proposed methodology arises from the comparison with existing standard proposals. Figure 7 shows a performance comparison of the matrix multiplication routine in a node with a multicore CPU and two different GPUs (`S0H1`) using different approaches:

- *Peak* is obtained for each problem size, n, by adding the performance obtained for this size with the MKL multiplication on CPU and the cuBLAS multiplication on each GPU. It represents the maximum achievable performance with these basic libraries, but the actual multiplications are carried out in each CU with smaller matrices (n_{gpu1}, n_{gpu2} and n_{cpu}, with $n = n_{gpu1} + n_{gpu2} + n_{\text{cpu}}$) and so the experimental results will not be close to this value.
- The performance using different searching methods for the AP values (*HL*, in green). HL_{Exh} corresponds to an exhaustive search on all the possible values of the parameters. The hierarchy is not exploited, and it represents a more realistic upper-bound than *Peak*. Two methods use the information of level 0, performing a search on the parameters of level 1 only (`S0H1-s0h0`): the exhaustive experi-

Table 5 Work division and performance of the successive matrix multiplications in an LU factorization $10{,}000 \times 10{,}000$ with block size 1000×1000, in ***12c_2C2075_4GTX590***

Matrix multiplication $n \times 1000 \times n$	Workload distribution	Performance
Size	*CPU,GPU0,GPU1,GPU2*	GFlops
9000	1980,1800,3420,1800	236
8000	1920,1600,3040,1440	377
7000	1540,1400,2660,1400	359
6000	1560,960,2280,1200	340
5000	1100,1000,1900,1000	320
4000	800,800,1600,800	292
3000	780,720,1500,0	273
2000	440,520,1040,0	211
1000	480,0,520,0	134

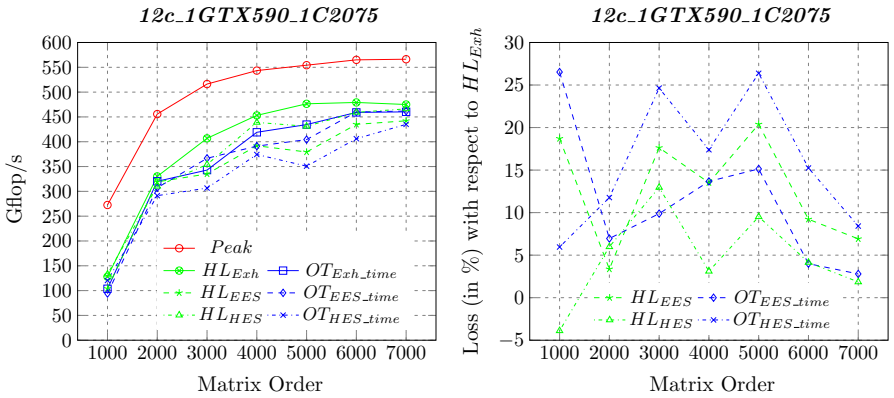

Fig. 7 Matrix multiplication routine in ***12c_1GTX590_1C2075***. Comparison of the performance obtained using three searching algorithms of the proposed hierarchical methodology (*HL*) and OpenTuner (*OT*) with the search time limit employed by each *HL* version. The results are the mean of ten executions for each matrix size. On the left, the performance in GFlops. On the right, the loss of performance of the methods exploiting the hierarchy and OpenTuner with the corresponding time limit with respect to the exhaustive search method

mental search (HL_{EES}) and the heuristic experimental search (HL_{HES}) described in Sect. 4.1. The total installation time employed for the problem sizes 1000, 2000, ..., 7000 was more than 3 days for HL_{Exh}, around 4 h for HL_{EES} and around 4 min for HL_{HES}. So, there is a great reduction in the search time with the hierarchical methodology, mainly when some heuristic is used.

- The performance obtained with OpenTuner (*OT*, in blue) where the search time has been limited to the search time of the three *HL* versions.

It can be observed how the performance obtained with each version of *HL* versus *OT* limited with the same search time are very similar, but slightly favorable to *HL*. The average loss of OpenTuner with respect to HL_{Exh} is 11.2%, and 15.7% with the time limits from HL_{EES} and HL_{HES}. The average losses of HL_{EES} and HL_{HES} are 12.8% and 4.8%, respectively. So, the systematic application of metaheuristics by OpenTuner gives slightly better results than those with the exhaustive search of the parameters at level 1, but a heuristic search on those parameters greatly improves the results with lower search times. This leads us to think that the integration of OpenTuner in the hierarchical approach as one of the possible methods to be used at some levels of the hierarchy could be interesting, and that neighborhood-based search methods should be considered to be included in OpenTuner.

7 Conclusions and future research

This work presents an autotuning methodology for parallel routines on heterogeneous platforms composed of hybrid nodes with a different number and type of processing elements (multicore and manycore). A hierarchical bottom-up

approach is proposed. The integration of the software and hardware hierarchies allows the complexity of this autotuning process to be addressed in a decentralized and modular manner. As proof of concept, the methodology is described using the matrix multiplication kernel like basic routine, and the extension to higher-level routines is discussed with the Strassen multiplication. The optimization of the basic multiplications is performed at the lowest level of the hierarchy for the underlying computing system, whereas the combination of those multiplications and their assignation to the computing units is decided at a second level. Therefore, the efficient exploitation of the computing units in the node is delegated to the underlying direct multiplication in the base case.

The Strassen multiplication is used as a case study to show the viability of the hierarchical autotuning methodology, but it can be applied similarly to other linear algebra routines, like matrix factorizations (LU, QR, Cholesky) implemented by blocks, with the most computationally demanding kernel being the matrix multiplication. So, the information generated when the basic matrix multiplication is installed at the different levels of a computing platform can be used for several higher-level routines, for which it is necessary to consider installations of the basic multiplication with sizes and shapes similar to those used in the routines in which it is going to be used. Furthermore, there are other routines of lower order which are called inside the higher-level routines. In the Strassen multiplication, these are additions and subtractions, and in matrix factorizations, they are normally non-block factorizations on smaller matrices and solutions of triangular systems also working on smaller matrices. For a particular routine, a graph of dependencies should be developed, and the tasks in the graph assigned to the computing units statically or dynamically. Even in linear algebra packages with dynamic assignation of tasks, the block or tile sizes need to be selected, so we are working on the adaptation of the methodology for libraries with dynamic assignation (e.g., Chameleon).

The hierarchical structure of the methodology makes it a generic approach. The search for appropriate values of the parameters for a problem size can be very costly, and so an exhaustive search (experimentation or theoretical estimation for all the possible values of the parameters, or for a representative set of values) can be substituted by a smarter search. OpenTuner is a powerful tool for parameters selection, and its integration in the hierarchical methodology is being considered. In any case, the use of more sophisticated search methods would contribute to reduce the installation time, but they do not change the validity of the hierarchical approach.

Acknowledgements This work was supported by the Spanish MCIU and AEI, as well as European Commission FEDER funds, under Grant RTI2018-098156-B-C53.

References

1. Agullo E, Demmel J, Dongarra J, Hadri B, Kurzak J, Langou J, Ltaief H, Luszczek P, Tomov S (2009) Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects. J Phys: Conf Ser 180(1):012037

2. Ansel J, Kamil S, Veeramachaneni K, Ragan-Kelley J, Bosboom J, O'Reilly U-M, Amarasinghe S (2014) OpenTuner: An extensible framework for program autotuning. In: 23rd International Conference on Parallel Architectures and Compilation Techniques. Edmonton, Canada, ACM, pp 303–316
3. Augonnet C, Thibault S, Namyst R, Wacrenier P-A (2011) StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurr Comput: Pract Exp 23(2):187–198
4. Batory D (1992) The design and implementation of hierarchical software systems with reusable components. ACM Trans Softw Eng Methodol 1:355–398
5. Bernabé G, Cuenca J, García L-P, Giménez D (2015) Auto-tuning techniques for linear algebra routines on hybrid platforms. J Comput Sci 10:299–310
6. Blackford LS, Choi J, Cleary A, D'Azevedo E, Demmel J, Dhillon I, Dongarra JJ, Hammarling S, Henry G, Petitet A, Stanley K, Walker D, Whaley RC (1997) ScaLAPACK user's guide. Society for Industrial and Applied Mathematics, Philadelphia
7. Cámara J, Cuenca J, Giménez D (2019) Hierarchical automatic optimization of high and medium level linear algebra routines. In: 18th International Conference on Computational and Mathematical Methods in Science and Engineering
8. Chameleon: Dense linear algebra subroutines for heterogeneous and distributed architectures. https://gitlab.inria.fr/solverstack/chameleon. Accessed Sept 2019
9. cuBLAS. http://docs.nvidia.com/cuda/cublas/. Accessed Sept 2019
10. Cuenca J, García L-P, Giménez D, Herrera F-J (2017) Guided installation of basic linear algebra routines in a cluster with manycore components. Concurr Comput: Pract Exp 29(15):e4112
11. Dackland K, Kågström B (1996) A hierarchical approach for performance analysis of ScaLAPACK-based routines using the distributed linear algebra machine. In: Applied Parallel Computing, Industrial Computation and Optimization, Third International Workshop, PARA96. Lyngby, Denmark, pp 186–195
12. Fatica M (2009) Accelerating Linpack with CUDA on heterogenous clusters. In: 2nd Workshop on General Purpose Processing on Graphics Processing Units. NY, USA, ACM, New York, pp 46–51
13. Golub G, Van Loan CF (2013) Matrix computations, 4th edn. The John Hopkins University Press, Baltimore
14. Goto K, van de Geijn RA (2008) Anatomy of high-performance matrix multiplication. ACM Trans Math Softw 34(3):12:1–12:25
15. Hasanov K, Quintin J-N, Lastovetsky AL (2015) Hierarchical approach to optimization of parallel matrix multiplication on large-scale platforms. J Supercomput 71(11):3991–4014
16. Intel MKL. http://software.intel.com/en-us/intel-mkl/. Accessed Sept 2019
17. Ohshima S, Kise K, Katagiri T, Yuba T (2007) Parallel processing of matrix multiplication in a CPU and GPU heterogeneous environment. In: 7th International Conference on High Performance Computing for Computational Science. Springer-Verlag, pp 305–318
18. Pfaffe P, Grosser T, Tillmann M (2019) Efficient hierarchical online-autotuning: A case study on polyhedral accelerator mapping. In: Proceedings of the ACM International Conference on Supercomputing, ICS '19, New York, USA, ACM, pp 354–366
19. PLASMA. http://icl.cs.utk.edu/plasma/. Accessed Sept 2019
20. Porterfield A, Bhalachandra S, Wang W, Fowler R (2016) Variability: a tuning headache. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp 1069–1072
21. Stanisic L, Thibault S, Legrand A, Videau B, Méhaut J-F (2015) Faithful performance prediction of a dynamic task-based runtime system for heterogeneous multi-core architectures. Concurr Comput: Pract Exp 27(16):4075–4090
22. Williams S, Oliker L, Carter J, Shalf J (2011) Extracting ultra-scale Lattice Boltzmann performance via hierarchical and distributed auto-tuning. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, New York, USA, ACM, pp 1–12
23. Yokota R, Barba L (2012) Hierarchical N-body simulations with autotuning for heterogeneous systems. Comput Sci Eng 14(3):30–39
24. Zhong Z, Rychkov V, Lastovetsky AL (2015) Data partitioning on multicore and multi-GPU platforms using functional performance models. IEEE Trans Comput 64(9):2506–2518