BLAS-3 Optimized by OmpSs Regions

(LASs Library)

Pedro Valero-Lara, Sandra Catalán
Barcelona Supercomputing Center (BSC)
Barcelona, Spain
{pedro.valero, sandra.catalan}@bsc.es

Xavier Martorell, Jesús Labarta
Universitat Politècnica de Catalunya
Barcelona, Spain
{xavim, jesus.labarta}@ac.upc.edu

***Abstract*—In this paper we propose a set of optimizations for the BLAS-3 routines of LASs library (Linear Algebra routines on OmpSs) and perform a detailed analysis of the impact of the proposed changes in terms of performance and execution time. OmpSs allows to use regions in the dependences of the tasks. This helps not only in the programming of the algorithmic optimizations, but also in the reduction of the execution time achieved by such optimizations. Different strategies are implemented in order to reduce the amount of tasks created (when there is enough parallelism) during the execution of BLAS-3 operations in the original LASs. Also a better IPC is obtained thanks to a better memory hierarchy exploitation. More specifically, we increase the performance, in particular on big matrices, about 12% for TRSM, and 17% for GEMM with respect to the original version of LASs, even using less cores in the case of GEMM/SYMM. Moreover, when LASs is compared to the OpenMP reference dense linear algebra library PLASMA, performance is increased up to 12.5% for GEMM/SYMM, while for TRSM/TRMM this value raises to 15%.**

***Index Terms*—BLAS-3, tasking, OmpSs, regions**

I. Introduction

Numerous scientific and engineering applications rely on linear algebra (LA) operations. The relevance of LA operations can be appreciated in a wide range of fields, such as determining the radar signature of a plane (e.g., Epsilon [1]), hyperspectral image processing (e.g., Opticks [2]), macromolecular simulations (e.g., GROMOS [3]), Computational Fluid Dynamics [4], [5], Image Processing [6], [7], just to mention a few. For this reason, the scientific community is constantly improving the performance of these LA operations. A proof of that is the existence of several (vendor and open-source) LA libraries and the constant development to improve performance [8]–[13], e.g. Intel MKL [14], IBM ESSL [15], PLASMA [16], libFLAME [17], Chameleon [18], ScaLAPACK [19], DPLASMA [20], NVIDIA Libraries (cuSparse, cuBLAS and cuSolver), among others.

All these libraries provide either BLAS [21] or LAPACK [22] functionality (or both) via different programming models that allow the parallelization of the code following different approaches. Depending on the programming model, the underlying strategy in order to exploit the hardware resources varies; some of them aim to make a better use of the memory hierarchy and, some others, focus on maximizing the use of all the available resources. Given that LAPACK routines often make use of BLAS level routines, a common target when trying to increase the performance of those libraries is to optimize BLAS level routines. In this vein, we present a set of optimizations on the BLAS-3 routines included in the novel open-source library for linear algebra operations called LASs (Linear Algebra routines on OmpSs) [1] and evaluate their potential benefits. In order to implement these optimizations we use *regions* [23], an advanced feature of OmpSs that allows to specify depencendes over a set of data without significant changes in the code. It is important to note that this feature is not part of the current OpenMP specification, so the optimizations and studies presented in this paper cannot be replicated using OpenMP. Other important point to highlight is that the porting of the implementations are not in need of any change to be used on other architectures.

The paper is structured as follows. Section II presents some related work, including several LA libraries that are based on different task-based programming models. Section III describes and evaluates the optimizations performed on BLAS-3 on the LASs library. Sections IV shows the conclusions and future work lines

II. State of the art

Today, we can find several examples of LA libraries implemented using task-based programming models. PLASMA provides parallel implementations of BLAS-3 and LAPACK level operations using OpenMP [24]. However, the initial parallelization was made using Quark [25]. Chameleon makes use of StarPU [26] runtime in order to implement dense linear algebra operations that can be run on shared and distributed memory scenarios.

In this paper we tackle the optimization of LASs, a LA library based on OmpSs [27]. OmpSs is a task-based programming model that extends OpenMP directives to give support to asynchronous parallelism and heterogeneity. We make use of the OmpSs programming model instead of others for the following reasons: i) This model presents an efficient management of the threads based on the use of queues, without the need of dealing with the overhead found in other models, such as the fork-join model used in OpenMP (see Figure 1). OmpSs also allows us to have a deeper control and analysis to evaluate the threads scheduling and the potential benefits of the optimizations proposed. ii) Unlike other programming models,

[1] https://pm.bsc.es/mathlibs/lass

DOI 10.1109/PDP.2019.00013

OmpSs is able to deal with regions in the dependences of the tasks efficiently. In fact, the effectiveness of the proposed optimizations present in this paper relies on this feature. iii) OmpSs is specially well integrated with the tools used for the performance evaluation Extrae and Paraver [28]. Extrae is a dynamic instrumentation package to trace programs compiled and run using OmpSs, OpenMP, pthreads, the message passing (MPI) programming model or a combination of these. Extrae generates trace files that can be later visualized with Paraver.

The rising in the amount of programming models based on task parallelism has been significant during the last decade, even making the OpenMP standard (initially designed for parallel regions) integrates this approach from version 3.0. Task-parallelism (see Figure 1) is an alternative paradigm that provides more flexibility, allowing the developer to deal with certain problems such as loops with an unknown length at run time.

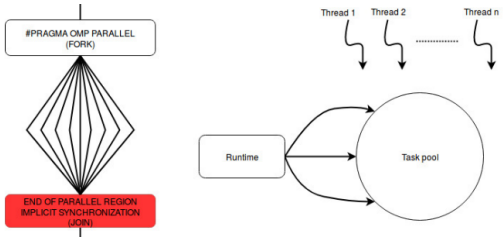

Fig. 1. OpenMP programming model vs. OmpSs.

III. BLAS-3 OPTIMIZATION

In this section, we explore a set of algorithmic optimizations on BLAS-3 level routines based on tasking and OmpSs region. We have divided the set of routines into three different subgroups according to their characteristics (scheduling, input/output parameters, parallelism): TRSM/TRMM, GEMM/SYMM, and SYRK/SYR2K.

The optimizations presented in this paper target the reduction of task instantiation time, the amount of tasks in the DAG generated by OmpSs and the exploitation of data locality, increasing the IPC, to achieve a reduction in the total execution time. A critical point regarding performance is the overhead introduced by the use of a runtime. For this reason, one of the objectives behind the presented optimizations implies the reduction of the amount of created tasks by joining small tasks into bigger ones when possible.

Additionally, when the number of tasks that need to be managed is reduced, data locality is better exploited. By creating bigger tasks a greater amount of data may be reused (if accessed properly) through the memory hierarchy, reducing execution time and thus increasing performance. This is done thanks to the OmpSs regions.

Load balancing is also considered when implementing our optimizations. OmpSs balances the workload automatically, however, the granularity of these tasks needs to be carefully determined in order to avoid relevant unbalance. For those operations where a critical unbalance has been detected, the solution lies on evenly distributing the workload among the available cores in the platform.

A. TRSM *and* TRMM *routines*

Both routines (TRSM and TRMM) have as input a triangular matrix and as output a regular dense matrix. The dependences between the different tiles are similar, so the same optimization can be applied to both routines.

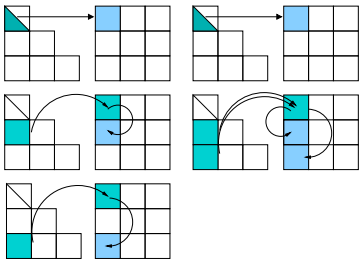

Fig. 2. TRSM implementation in LASs (left) and LASs-opt (right), respectively.

Figure 2 shows the tiled algorithm for TRSM (left) when using one task per tile and the optimization implemented in LASs-opt (right). The optimization consists of joining the set of GEMM, which compute the tiles of one tiled-column, instead of using one task per tile. In this case, we can reduce the number of tasks to be computed, as shown in Figure 3 and, consequently, reduce the overhead of task management.

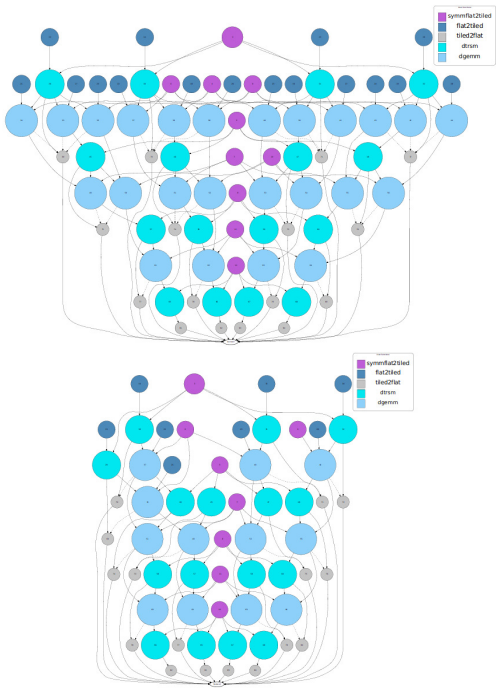

Fig. 3. TRSM DAG in LASs using one task-dgemm per tile (top) and LASs-opt using one task computing many dgemm per tiled-column (bottom), respectively. The dark-blue and violet nodes correspond to the tasks which compute the data-layout transformations, the light-blue and blue nodes correspond to the TRSM and GEMM tasks respectively.

Listing 1. Code for TRSM in LASs.

```
for ( d = 0; d < dt; d++) {
   for ( c = 0; c < ct; c++) {
      #pragma oss task in(TILE_A[d][d])
                       inout(TILE_B[d][c])
                       shared(TILE_A, TILE_B)
                       firstprivate(d, c)
      dtrsm( ... );
   }
   for ( r = 0; r < rt; r++)  {
      for ( c = 0; c < ct; c++) {
         #pragma oss task in(TILE_A[d][r])
                          in(TILE_B[d][c])
                          inout(TILE_B[r][c])
                          shared(TILE_A, TILE_B)
                          firstprivate(d, r, c)
         dgemm( ... );
      }
   }
}
```

This optimization is not in need of big modifications in the code thanks to OmpSs region mecanism. Basically, it requires to move the task instantiation and the modification of the data-dependence clauses, as can be seen in Listing 1 and Listing 2. After evaluating different approaches and task granularity, the best configuration consists of using two tasks per tiled-column in the computation of the set of GEMM. As shown, to perform this optimization we make use of regions in the instantiation of GEMM tasks.

To carry out this optimization efficiently, it is also necessary to have as many tiled-columns (`ct` in Listing 2) as number of cores. Due to this, we need to change the tile size according to the next equation:

$$tile\_size = N/\#cores\ available$$

Being N the number of columns of matrix B. For example, for a given problem of N (number of columns of matrix B) and M (number of rows of matrix B) = 24,576, on a platform where $\#cores = 48$, the tile size must be equal to $512^2, 512 = 24,576/48$ [2].

All the tests in this section (and in the rest of this paper) have been performed on one node of the Mare Nostrum 4 supercomputer, featuring two sockets Intel Xeon Platinum 8160 with 24 cores each at 2.10GHz with a total amount of 48 cores. The maximum performance per node is 2,300 GFLOPS [29]. Moreover, each core has 32KB of L1 data cache, 32KB of L1 instructions cache 32K, and 1MB of L2 cache. All the cores in the same socket share an L3 cache of 33MB and the main memory is 96GB. In addition, in all executions we make use of `numactl --interleave=all` to equally distribute the data used between the sockets memory. All the computations are performed in double precision. Moreover, performance results for PLASMA, as a reference dense linear algebra library implemented in OpenMP, are presented in all cases. Both libraries, LASs and PLASMA, make use of the Intel MKL library to compute the single threaded BLAS-3 routines (see Listing 1 and Listing 2).

[2]Note that the default tile size of LASs and PLASMA is 512^2 and 256^2 respectively

Listing 2. Code for TRSM in LASs-opt

```
for ( d = 0; d < dt; d++) {
   for ( c = 0; c < ct; c++) {
      #pragma oss task in(TILE_A[d][d])
                       inout(TILE_B[d][c])
                       shared(TILE_A,TILE_B)
                       firstprivate(d, c)
      dtrsm( ... );
   }
   for ( r = 0; r < rt; r+=2)  {
      #pragma oss task in(TILE_A[d][r])
                       in(TILE_B[d][0:ct-1])
                       inout(TILE_B[r][0:ct-1])
                       shared(TILE_A,TILE_B)
                       firstprivate(d, r, c)
      for ( c = 0; c < ct; c++) {
         dgemm( ... );
      }
   }

   for ( r = 1; r < rt; r+=2)  {
      #pragma oss task in(TILE_A[d][r])
                       in(TILE_B[d][1:ct-1])
                       inout(TILE_B[r][1:ct-1])
                       shared(TILE_A,TILE_B)
                       firstprivate(d, r, c)
      for ( c = 0; c < ct; c++) {
         dgemm( ... );
      }
   }
}
```

As we see in the results (Figures 4 and 5), this has important consequences on performance. In both cases, it is shown that the optimized implementation of LASs attains higher performance for big matrices (above $24,576^2$ elements for TRSM and $18,432^2$ elements for TRMM). However, this optimization turns out to be inefficient when dealing with medium or relative small matrices, since small matrices imply a small tile size, and so a low IPC on GEMM computation, being, in these cases, the original LASs and PLASMA faster. On the contrary, for the biggest analyzed matrix, the IPC goes from 1.67 (average), 1.70 (maximum), and 1.60 (minimum) for the original LASs to 2.02 (average), 2.15 (maximum), and 1.84 (minimum) for the optimized implementation of LASs. The IPC achieved by PLASMA, LASs and LASs-opt are reported in Table III.

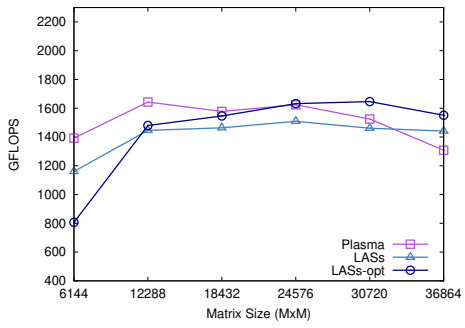

Fig. 4. Performance of trsm.

This increase in performance, up to 12% and 15% for TRSM with respect to PLASMA and LASs, respectively, and up to

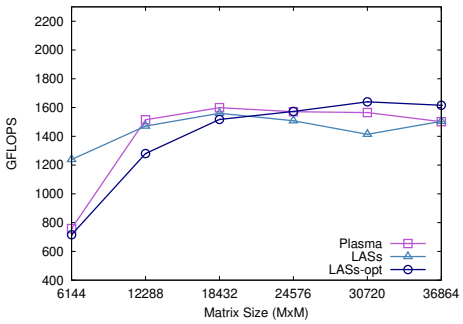

Fig. 5. Performance of trmm.

TABLE I
EXECUTION TIME FOR TRSM.

Lib. / size	6144	12288	18432	24576	30720	36864
PLASMA	**0.16**	**1.12**	**3.96**	**9.20**	19.89	38.29
LASs	0.20	1.45	4.28	9.83	19.83	36.61
LASs-opt	0.29	1.50	4.06	9.44	**17.45**	**32.29**

7% and 15% for TRMM for the same scenarios, is due to the fact that several GEMM are now joined in the same task in order to compute a column instead of a tile. Using this approach, the runtime overhead is reduced, since less tasks are created and so, a smaller DAG is created. Moreover, a better exploitation of the memory hierarchy is made thanks to the increase in data locality.

The impact of this optimization for TRSM is shown in Figure 6, where a substantial reduction in execution time is observed when applying this change to the biggest matrix used (up to 6 seconds w.r.t. PLASMA for TRSM on big matrices). It is remarkable the reduction of the creation time of the tasks, which changes from 11.11 seconds to 3.9 seconds. This is clearly seen in the traces, where the creation tasks (in red for LASs and dark-green for LASs-opt at the beginning of the execution of the first core) take much longer in the case of the original LASs implementation.

The execution time, percentage of execution time, number of tasks, and IPC for each type of task for the presented traces is summarized in Table III. These results show that the most computationally expensive part of the operation are GEMM tasks; in fact, this is the part where the optimizations of LASs-opt provide the higher reduction in execution time as well, compensating the increase in execution time that is observed in data-layout transformation (dlt) and TRSM tasks compared to original LASs. PLASMA presents the biggest time for dlt and GEMM, but the shortest time for TRSM.

In terms of number of TRSM and GEMM tasks (see Table III)

TABLE II
EXECUTION TIME FOR TRMM.

Lib. / size	6144	12288	18432	24576	30720	36864
PLASMA	0.30	**1.22**	**3.91**	9.51	18.60	33.33
LASs	**0.19**	1.44	4.01	9.84	20.50	33.29
LASs-opt	0.33	1.64	4.13	**9.43**	**17.67**	**30.98**

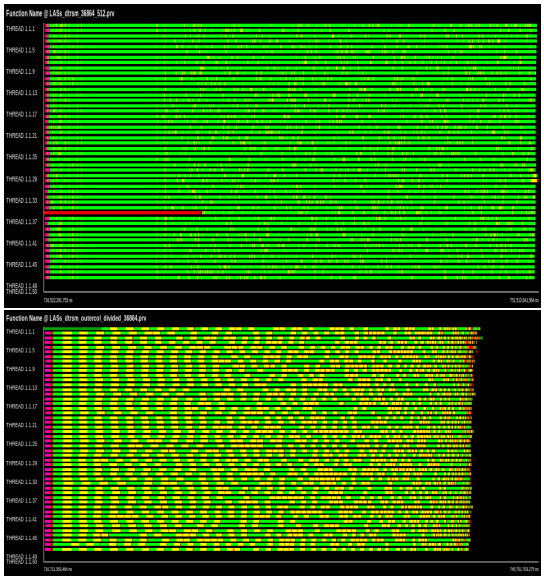

Fig. 6. Trace of TRSM ($n = 36864, b = 512$) with the initial implementation of LASs (top) and the optimized version of the operation (bottom).

TABLE III
%, TIME, IPC AND #TASKS CONSUMED PER TASK IN TRSM

PLASMA	dlt	trsm	gemm
%	4.03	0.83	95.14
Time (ns)	6.8×10^{10}	1.3×10^{10}	1.6×10^{12}
IPC	0.01	1.40	2.24
#Tasks	61,090	20,736	1,482,624
LASs	**dlt**	**trsm**	**gemm**
%	2.12	1.44	95.78
Time (ns)	3.3×10^{10}	2.35×10^{10}	1.56×10^{12}
IPC	1.10	1.68	1.70
#Tasks	12,996	5,184	184,032
LASs-opt	**dlt**	**trsm**	**gemm**
%	2.23	2.18	95.34
Time (ns)	3.16×10^{10}	3.15×10^{10}	1.37×10^{12}
IPC	1.10	1.97	2.05
#Tasks	5,184	2,307	4,615

the number of tasks needed in LASs-opt is considerably lower than in the case of LASs and much lower if compared to PLASMA. With respect to LASs around 50% less tasks are created for data-layout transformations and TRSM, using only about 2% of GEMM tasks computed by LASs. Regarding PLASMA, the number of tasks when using LASs-opt is about 92% less for data-layout transformations, 89% for TRSM and 99.5% for GEMM.

Regarding IPC, the optimizations implemented on LASs suppose an important increment w.r.t. original LASs in all the tasks except for the dlt tasks. In PLASMA, the IPC achieved by dlt is very low. The TRSM tasks, although much better than dlt, are still lower than original LASs and LASs-opt. However, the IPC obtained by GEMM tasks is slightly bigger than in LASs-opt. Despite this better IPC on GEMM tasks, LASs-opt is able to reduce the time consumed by these tasks about 15%, because of a much lower number of tasks and a better memory exploitation in general.

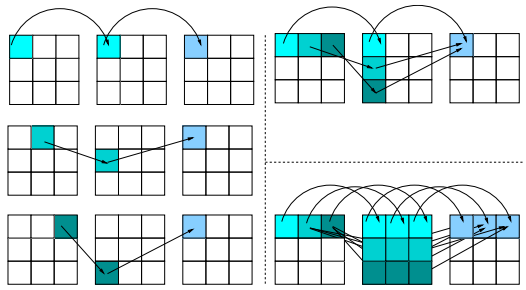

Fig. 7. GEMM implementation in LASs (left), GEMM implementation in LASs-opt-app1 (top right), and GEMM implementation in LASs-opt-app2 (bottom right).

B. GEMM *and* SYMM *routines*

In this section we explore some optimizations on GEMM and SYMM. Both routines make use of three dense matrices as parameters, two inputs and one output. The memory access pattern and dependences, although not so similar as in the previous case between TRSM and TRMM, are still similar in GEMM and SYMM. The major difference is that in SYMM a triangular matrix A is used.

In these routines, we have explored three different variants as presented in Figure 7: one task per GEMM (left), one task per tile of C (top right) and one task per row of C (bottom right). The corresponding codes that implement these three variants are listed in Listings 3 to 5.

Listing 3. Code for GEMM in LASs

```
1  for ( m = 0; m < mt; m++){
2      for ( n = 0; n < nt; n++) {
3          for ( k = 0; k < kt; k++)  {
4              #pragma oss task
5                       in(TILE_A[m][k])
6                       in(TILE_B[k][n])
7                       inout(TILE_C[m][n])
8                       shared(TILE_A,TILE_B,
                            TILE_C)
9                       firstprivate(m, n, k)
10             dgemm( ... );
11         }
12     }
13 }
```

Listing 4. Code for GEMM in LASs-opt (one task per tile of C).

```
1  for ( m = 0; m < mt; m++){
2      for ( n = 0; n < nt; n++) {
3          #pragma oss task
4                   in(TILE_A[m][0:kt-1])
5                   in(TILE_B[0:kt-1][n])
6                   inout(TILE_C[m][n])
7                   shared(TILE_A,TILE_B,TILE_C)
8                   firstprivate(m, n)
9          for ( k = 0; k < kt; k++) {
10             dgemm( ... );
11         }
12     }
13 }
```

Listing 5. Code for GEMM in LASs-opt (one task per row).

```
1  for ( m = 0; m < mt; m++){
2      #pragma oss task
3               in(TILE_A[m][0:kt-1])
4               in(TILE_B[0:kt-1][0:nt-1])
5               inout(TILE_C[m][0:nt-1])
6               shared(TILE_A,TILE_B,TILE_C)
7               firstprivate(m)
8      for ( n = 0; n < nt; n++) {
9          for ( k = 0; k < kt; k++) {
10             dgemm( ... );
11         }
12     }
13 }
```

Depending on the approach, a different tile size is used. For the first approach (one task per GEMM), we use a tile size equal to 512^2 (default tile size in LASs). The tile size in the second (LASs-opt-app1) and third (LASs-opt-app2) approaches depends on the problem size. For an efficient execution of the second approach (one task per tile of matrix C), we must have as many tasks as number of cores. This forces to use a big tile size. In the third approach, we use the same strategy used in TRSM, that is, the size of the tile must be equal to $M/\#cores$, being M the number of rows of matrix C, so for a problem size equal to $36,864^2$, we need a tile size equal to 768^2.

To evaluate each of the approaches, we first analyze the traces of each implementation on the biggest matrix used ($30,720^2$ elements). Figure 8 presents the execution trace when using one task per GEMM (top), one task per tile of C (middle) and one task per row of C (bottom).

In the first approach (one task per GEMM), the time (in green) consumed by the instantiation of the tasks is the largest. This is an expected behavior, as this approach makes use of the largest number of tasks with respect to the other approaches. In the second approach (one task per tile of matrix C), although a lower instantiation time is required, it is not possible to homogeneously distribute the tiles among the cores given that our test platform has 48 cores. In this scenario $sqrt(48) = 6.92$, being not a natural number, and this causes an important unbalance. Finally, the third approach (one task per row of C) reduces considerably the time consumed by the instantiation of the tasks, however, it is not able to overlap execution with data-layout transformation, since until the whole row of C is computed the transformation cannot be done. This causes also a non-negligible unbalance.

As an overview, we show the performance in GFLOPS (Figures 9 and 10) and execution time in seconds (Tables IV and V) for GEMM and SYMM. In each case we analyze the behavior for LASs (one task per operation), LASs-opt-app2 (one task per row of matrix C) and PLASMA.

As shown in the execution time results, LASs-opt-app2 approach is only faster than the original LASs and PLASMA when using the biggest matrix size, reducing the execution time 5% for GEMM and 7% for SYMM.

Analysis of the "one task per tile of C" approach: Finally, and in order to evaluate the efficiency of the second approach (one task per tile of matrix C) more deeply, we include one

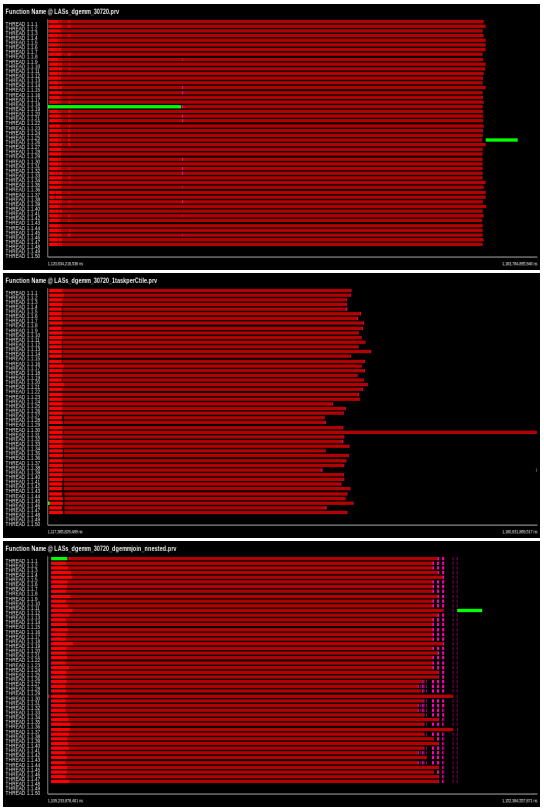

Fig. 8. Traces of GEMM.

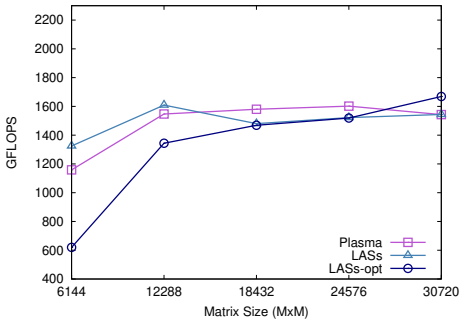

Fig. 9. Performance of gemm.

additional analysis using only 36 cores of the 48 available in our test platform, against the original LASs using 48 cores. To this end, we distribute the computation using the first 18 cores of each socket. In Figure 11, we show the traces of original LASs on 48 cores (top) and the second approach (LASs-opt-app2) on a matrix size equal to $30,720^2$. In the original LASs

TABLE IV
EXECUTION TIME FOR GEMM IN LASS AND LASS-OPT IMPLEMENTATIONS.

Time(s)	6144	12288	18432	24576	30720
PLASMA	0.40	**2.39**	**7.92**	**18.62**	37.54
LASs	**0.35**	2.61	8.46	19.50	37.54
LASs-opt-app2	0.58	3.47	8.39	23.68	**35.81**

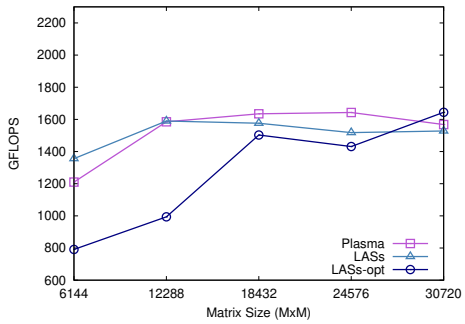

Fig. 10. Performance of symm.

TABLE V
EXECUTION TIME FOR SYMM IN LASS AND LASS-OPT IMPLEMENTATIONS.

Time(s)	6144	12288	18432	24576	30720
PLASMA	0.38	**2.34**	**7.65**	**18.06**	36.97
LASs	**0.34**	2.64	7.94	19.55	37.94
LASs-opt-app2	0.59	4.22	8.33	20.73	**35.27**

the tile size is equal to 512^2 (default tile size in LASs) and in the second approach (using 36 cores), taking into account that the same number of tasks and cores is required, the tile size is equal to $5,120^2$.

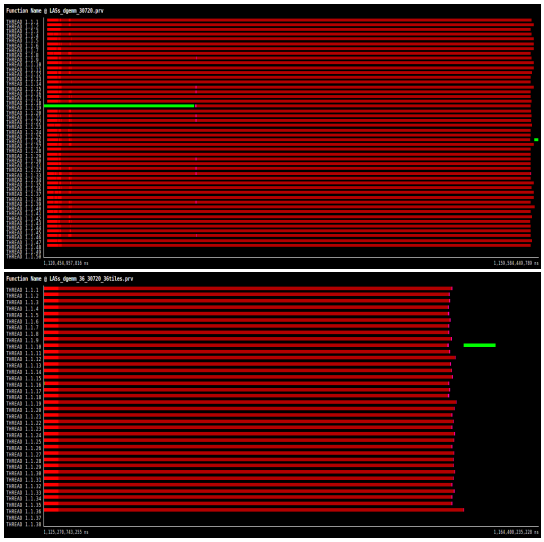

Fig. 11. Traces of GEMM for LASs implementation with 48 cores (top) and LASs-opt with 36 cores (bottom).

As shown in the traces, we see an important reduction of the execution time, even using a lower number of cores. To continue analyzing this approach, we extend the analysis showing the GFLOPS (Figure 12) and execution time (Table VI) of this approach (using 36 cores) against the original LASs and PLASMA using 48 cores.

Results presented for performance and time execution show that the LASs-opt-app2 approach is able to outperform original LASs and PLASMA with a peak reduction time of 17% and 12.5%, respectively, using less cores.

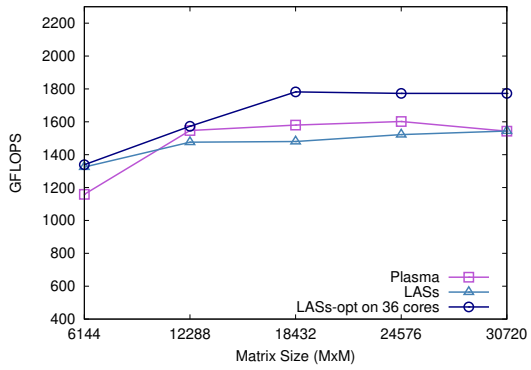

Fig. 12. Performance of GEMM for LASs implementation with 48 cores and LASs-opt with 36 cores.

TABLE VI
EXECUTION TIME FOR LASS AND LAS-OPT (ONE TASK PER TILE OF MATRIX C) WITH 48 AND 36 CORES RESPECTIVELY.

Time(s)	6144	12288	18432	24576	30720
PLASMA	0.40	2.39	7.92	18.62	37.56
LASs	**0.35**	2.51	8.46	19.50	37.54
LASs-opt	**0.35**	**2.36**	**7.03**	**16.74**	**33.22**

C. SYRK *and* SYR2K *routines*

Following the same idea as for the case of TRSM and TRMM, SYRK and SYR2K have been optimized applying the same approach, joining several GEMM tasks into a bigger one and using the appropriate tile size depending on the number of cores in the platform. We consider the same optimizations for SYRK and SYR2K because they present important similarities in terms of input/output parameters as well as in terms of data dependences and task scheduling. The major difference found is that in SYR2K, one more dense matrix B is involved in the execution.

In both routines we can find the same scenario graphically illustrated in Figure 13. This figure shows that the GEMM computed in these routines follows the same pattern; GEMM whose inputs come from different columns store the result in the same output tile. This makes difficult to follow the strategy used in the previous routines, which provokes blocking between different tasks (columns) when joining GEMM tasks of the same column into a bigger task sequentializing the execution.

The described behavior is clearly shown in the traces of Figure 14. When comparing the original LASs (top) with the optimization (bottom), we can observe how the blocking among the tasks (gaps in the trace) increases the execution time. In this scenario, joining several GEMM tasks into a bigger one reduces performance drastically.

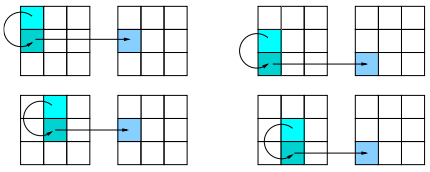

Fig. 13. SYRK task scheduling.

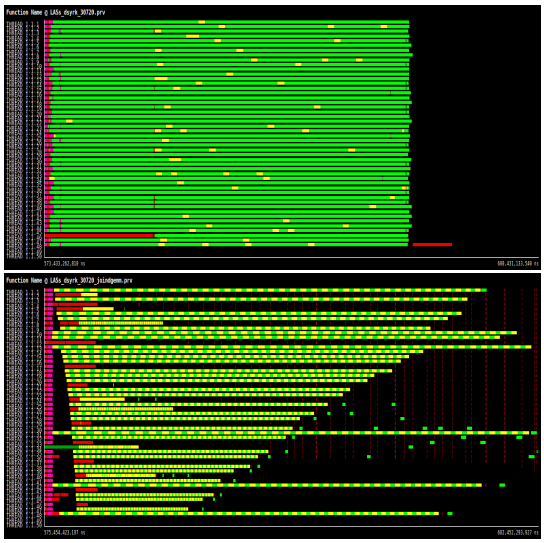

Fig. 14. Trace of SYRK execution using LASs (top) and LASs-opt (bottom) implementations.

As in the previous cases, we also include performance results in terms of GLFOPS for SYRK and SYR2K. Figures 15 and 16 report the results obtained for original LASs, LASs-opt and PLASMA for SYRK and SYR2K, respectively. In the light of these results, unlike the other routines, the optimization based on joining several GEMM tasks into one bigger task is not effective on SYRK and SYR2K. However, the original LASs matches the performance attained by PLASMA, even improving it for small matrices.

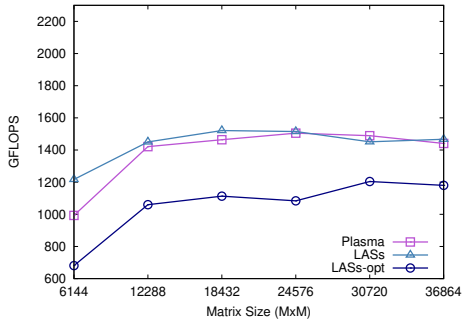

Fig. 15. Performance of syrk.

IV. CONCLUSIONS AND FUTURE WORK

The proposed optimizations based on the use of OmpSs regions are able to improve the performance of most of the considered BLAS-3 operations for big matrices (except for SYRK and SYR2K). The reason behind this behavior is the fact that the time needed to create the tasks is now shorter thanks to the reduction in the amount of tasks created; moreover, and more important, the IPC is increased due to a better memory

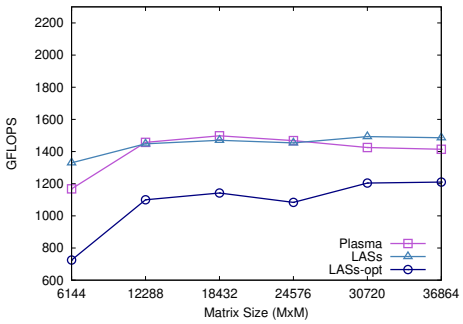

Fig. 16. Performance of syr2k.

management when several GEMM tasks are joined into a bigger GEMM.

We conclude that it is important to be able to to decide the best strategy depending on the input matrix size not only to improve BLAS level routines performance, but also LAPACK operations. For this reason, As future work we plan to implement LAPACK representative operations relying on the optimized version of LASs.

ACKNOWLEDGMENT

This project has received funding from the Spanish Ministry of Economy and Competitiveness under the project Computación de Altas Prestaciones VII (TIN2015-65316-P), the Departament d'Innovació, Universitats i Empresa de la Generalitat de Catalunya, under project MPEXPAR: Models de Programació i Entorns d'Execució Parallels (2014-SGR-1051), and the Juan de la Cierva Grant Agreement No IJCI-2017-33511. We also acknowledge the funding provided by Fujitsu under the BSC-Fujitsu joint project: Math Libraries Migration and Optimization.

REFERENCES

[1] "Epsilon." [Online]. Available: https://www.roke.co.uk/ /media/Files/C/Chemring-Roke/documents/01331-Epsilon.pdf

[2] N. Jennings, "Opticks open source remote sensing and image processing software, a community college gis program, and collaboration," *OSGeo Journal*, vol. 10, no. 1, p. 5, 2012.

[3] M. Christen, P. H. Hünenberger, D. Bakowies, R. Baron, R. Bürgi, D. P. Geerke, T. N. Heinz, M. A. Kastenholz, V. Kräutler, C. Oostenbrink *et al.*, "The gromos software for biomolecular simulation: Gromos05," *Journal of computational chemistry*, vol. 26, no. 16, pp. 1719–1751, 2005.

[4] P. Valero-Lara, A. Pinelli, J. Favier, and M. Prieto-Matías, "Block tridiagonal solvers on heterogeneous architectures," in *10th IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA 2012, Leganes, Madrid, Spain, July 10-13, 2012*, 2012, pp. 609–616.

[5] P. Valero-Lara, A. Pinelli, and M. Prieto-Matías, "Fast finite difference poisson solvers on heterogeneous architectures," *Computer Physics Communications*, vol. 185, no. 4, pp. 1265–1272, 2014.

[6] P. Valero-Lara, "A GPU approach for accelerating 3d deformable registration (DARTEL) on brain biomedical images," in *20th European MPI Users's Group Meeting, EuroMPI '13, Madrid, Spain - September 15 - 18, 2013*, 2013, pp. 187–192.

[7] ——, "Multi-gpu acceleration of DARTEL (early detection of alzheimer)," in *2014 IEEE International Conference on Cluster Computing, CLUSTER 2014, Madrid, Spain, September 22-26, 2014*, 2014, pp. 346–354.

[8] J. J. Dongarra, S. Hammarling, N. J. Higham, S. D. Relton, P. Valero-Lara, and M. Zounon, "The design and performance of batched BLAS on modern high-performance computing systems," in *International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland*, 2017, pp. 495–504.

[9] P. Valero-Lara, I. Martínez-Peréz, A. J. Peña, X. Martorell, R. Sirvent, and J. Labarta, "cuhinesbatch: Solving multiple hines systems on gpus human brain project*," in *International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland*, 2017, pp. 566–575.

[10] P. Valero-Lara, I. Martínez-Peréz, S. Mateo, R. Sirvent, V. Beltran, X. Martorell, and J. Labarta, "Variable batched DGEMM," in *26th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP 2018, Cambridge, United Kingdom, March 21-23, 2018*, 2018, pp. 363–367.

[11] P. Valero-Lara, I. Martínez-Pérez, R. Sirvent, X. Martorell, and A. J. Peña, "cuthomasbatch and cuthomasvbatch, CUDA routines to compute batch of tridiagonal systems on NVIDIA gpus," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 24, 2018.

[12] P. Valero-Lara, R. Sirvent, A. J. Peña, X. Martorell, and J. Labarta, "Mpi+openmp tasking scalability for the simulation of the human brain: Human brain project," in *Proceedings of the 25th European MPI Users' Group Meeting, Barcelona, Spain, September 23-26, 2018*, 2018, pp. 5:1–5:8.

[13] Valero-Lara, Pedro, Martínez-Pérez, Ivan, Sirvent, Raül, Peña, Antonio J., Martorell, Xavier, and Labarta, Jesús, "Simulating the behavior of the human brain on gpus," *Oil Gas Sci. Technol. - Rev. IFP Energies nouvelles*, vol. 73, p. 63, 2018. [Online]. Available: https://doi.org/10.2516/ogst/2018061

[14] Intel Corp., "Intel math kernel library (MKL) 11.0," http://software.intel.com/en-us/intel-mkl.

[15] IBM, "Engineering and Scientific Subroutine Library," http://www.ibm.com/systems/software/essl/, 2012.

[16] "PLASMA project home page," http://icl.cs.utk.edu/plasma.

[17] F. G. Van Zee, `libflame`*: The Complete Reference*. `www.lulu.com`, 2009.

[18] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. P. Thibault, "Achieving high performance on supercomputers with a sequential task-based programming model," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1–1, 2017.

[19] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1997.

[20] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemariner, H. Ltaeif, P. Luszczek, A. YarKhan, and J. Dongarra, "Flexible development of dense linear algebra algorithms on massively parallel architectures with dplasma." Anchorage, Alaska, USA: IEEE, 05-2011 2011, pp. 1432–1441.

[21] Netlib.org, "Blas," http://www.netlib.org/blas.

[22] ——, "Lapack," http://www.netlib.org/lapack.

[23] J. M. Pérez, R. M. Badia, and J. Labarta, "Handling task dependencies under strided and aliased references," in *Proceedings of the 24th International Conference on Supercomputing, 2010, Tsukuba, Ibaraki, Japan, June 2-4, 2010*, 2010, pp. 263–274.

[24] "OpenMP API for parallel programming, version 4.5," https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[25] A. YarKhan, J. Kurzak, and J. Dongarra, "Quark users' guide: Queueing and runtime for kernels," Tech. Rep. ICL-UT-11-02, 00-2011 2011.

[26] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, vol. 23, pp. 187–198, Feb. 2011. [Online]. Available: http://hal.inria.fr/inria-00550877

[27] "OmpSs project home page," http://pm.bsc.es/ompss.

[28] G. Llort, H. Servat, J. Gonzlez, J. Gimnez, and J. Labarta, "On the usefulness of object tracking techniques in performance analysis," in *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2013, pp. 1–11.

[29] "RES - Red Espaola de Supercomputacin," https://www.res.es/en/res-sites/marenostrum-minotauro.