Towards an Auto-tuned and Task-based SpMV (LASs Library) *

Sandra Catalán[1], Tetsuzo Usui[2], Leonel Toledo[3] Xavier Martorell[4], Jesús Labarta[4], and Pedro Valero-Lara[3]

[1] Universidad Complutense de Madrid (UCM), Madrid, Spain
[2] Next Generation Technical Computing Unit, Fujitsu Limited, Kawasaki, Japan
[3] Barcelona Supercomputing Center (BSC), Barcelona, Spain
[4] Universitat Politècnica de Catalunya Barcelona, Spain

Abstract. We present a novel approach to parallelize the SpMV kernel included in LASs (Linear Algebra routines on OmpSs) library, after a deep review and analysis of several well-known approaches. LASs is based on OmpSs, a task-based runtime that extends OpenMP directives, providing more flexibility to apply new strategies. Based on tasking and nesting, with the aim of improving the workload imbalance inherent to the SpMV operation, we present a strategy especially useful for highly imbalanced input matrices. In this approach, the number of created tasks is dynamically decided in order to maximize the use of the resources of the platform. Throughout this paper, SpMV behavior depending on the selected strategy (state of the art and proposed strategies) is deeply analyzed, setting in this way the base for a future auto-tunable code that is able to select the most suitable approach depending on the input matrix. The experiments of this work were carried out for a set of 12 matrices from the Suite Sparse Matrix Collection, all of them with different characteristics regarding their sparsity. The experiments of this work were performed on a node of Marenostrum 4 supercomputer (with two sockets Intel Xeon, 24 cores each) and on a node of Dibona cluster (using one ARM ThunderX2 socket with 32 cores). Our tests show that, for Intel Xeon, the best parallelization strategy reduces the execution time of the reference MKL multi-threaded version up to 67%. On ARM ThunderX2, the reduction is up to 56% with respect to the OmpSs parallel reference.

Keywords: SpMV · Parallel Programming · Tasking · Auto-tuning · Taskloop · Nesting · LASs · OmpSs.

* 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, and the Spanish Ministry of Science and Innovation under the project Heterogeneidad y especialización en la era post-Moore (RTI2018-093684-B-I00). We also acknowledge the funding provided by Fujitsu under the BSC-Fujitsu joint project: Math Libraries Migration and Optimization.

Catalán, S. [et al.]. Towards an auto-tuned and task-based SpMV (LASs Library). A: International Workshop on OpenMP. "OpenMP: Portable Multi-Level Parallelism on Modern Systems, 16th International Workshop on OpenMP, IWOMP 2020: Austin, TX, USA, September 22–24, 2020: proceedings". Berlín: Springer, 2020, p. 115-129. ISBN 978-3-030-58144-2. DOI 10.1007/978-3-030-58144-2_8.
The final authenticated version is available online at https://doi.org/10.1007/978-3-030-58144-2_8

1 Introduction

Sparse linear algebra is key in many scientific and engineering applications. One of the most representative and used operations is the sparse matrix-vector product (SpMV), defined as

$$y := \alpha Ax + \beta y, \tag{1}$$

where α and β are scalars, x and y are dense vectors and, A is a sparse matrix. The sparse nature of the input matrix makes this operation highly unbalanced, due to the non-uniform pattern when accessing the elements of the matrix. However, several storage formats have been proposed in order to palliate this effect.

The relevance of SpMV kernel is shown in the wide range of vendors and open-source libraries [9, 3, 6, 1], and the large number of applications that make use of it. A few of these reference sparse linear algebra libraries are MUMPS [4], that implements a parallel sparse direct solver, SuperLU [12], a general purpose library for the direct solution of systems of linear equations, MAGMA-Sparse [5], that provides sparse linear algebra solutions for heterogeneous architectures, cuSparse [1], which contains a set of basic sparse linear algebra subroutines developed by Nvidia, PETSC [6], a suite of data structures and routines for the solution of partial differential equations, FenicS [3], an open-source computing platform for solving partial differential equations, or HPCG [9], a benchmark project that aims to create a new metric for ranking HPC systems.

In this work, we focus on the sparse matrix-vector kernel (kdspmv) in LASs [5], a linear algebra library based on OmpSs [2, 24, 23]. Given that LASs is implemented in OmpSs, the analyzed strategies are implemented with this programming model throughout this work, although other programming models can be used to this end and benefit from those approaches. OmpSs is an open-source programming model [10] that has the following advantages in contrast to other runtimes: i) The model presents efficient management of the threads based on the use of queues, without the need of dealing with the overhead found in others models, such as the fork-join model used in OpenMP. ii) OmpSs is specifically designed for the use of tasks, making it a good choice for the study of task-based approaches. iii) It allows the user to have deeper control of the thread scheduling. iv) It provides us with tighter control and better knowledge about the taskloop implementation necessary to improve the proposed optimizations of the code, especially for nesting. iv) OmpSs is especially well integrated with the tools used for performance evaluation Extrae and Paraver. Extrae [13] is a dynamic instrumentation package to trace programs which generates trace files that can be later visualized with Paraver. Thanks to the its integration with OmpSs more information can be retrieved for those implementations in comparison to other programming models.

We propose and analyze different strategies in order to parallelize the SpMV kernel included in LASs library [18, 19, 17], which implements the general SpMV (see Equation 1) and operates on an input matrix stored in CSR format [11]. The main challenge we target through the parallelization of this kernel is balancing

[5] https://pm.bsc.es/gitlab/pvalero/lass/

the computations among the cores in order to attain good performance. Four different parallel approaches based on OmpSs features are proposed and analyzed to tackle sparsity and achieve a balanced workload distribution.

2 State of the art

Sparse matrices are present in a wide variety of applications used in very different fields such as graph analytics or economics. All these applications require the resolution of large-scale linear systems, usually done through iterative methods, and/or eigenvalue problems, whose most relevant component is the SpMV. For this reason, improving the portability of this kernel and increasing the performance delivered by making good use of the underlying resources is key for the mentioned applications.

Big efforts have been carried out by the scientific community in order to increase SpMV performance. An important part of the optimization of scientific codes consists of using the appropriate format to represent matrices in memory [21, 20, 8, 25]. Following different approaches, cache performance, data locality and, consequently, the overall performance of SpMV, has been proven to be affected substantially. Some of the most common formats for sparse matrices are Coordinate format (COO), Compressed Row Storage (CRS), Compressed Column Storage (CCS) [11] or ELLPACK-R [15]. Among these options, CSR is the most widely used and the de facto standard due to the fact that no assumptions on the sparsity structure of the matrix are made.

There exist several works that target the parallelization of SpMV on multi-core CPU, GPU, and MIC (many integrated cores). In [14] different scheduling strategies for particular matrices are explored for both architectures, multi-core CPU (SPARC64 IXfx and Intel Xeon Ivy Bridge-EP) and MIC (Knights Corner). Following the same type of comparison, but focused on analyzing the impact of using a hybrid MPI/OpenMP approach to make better exploitation of the hardware resources, [26] presents the results on the Knights Corner. Halfway between applying new parallelization algorithms and choosing an appropriate storage format, in [27] the authors propose the Blocked Compressed Common Coordinate (BCCOO) storage format and improve load balancing through a matrix-based segmented sum/scan algorithm on AMD FirePro W8000, GeForce Titan X, and Nvidia Tesla K20.

The analysis of the bibliography regarding SpMV shows that works in this area mostly focus on studying and proposing new storage formats that exploit better the features of specific hardware or application. On the contrary, in this work, we focus on CSR format, the most wide-used format for sparse matrices, and target algorithms that can be easily implemented and tuned on a multi-core CPU.

3 Parallelizing SpMV

Parallelizing SpMV is key to solve nowadays problems in a wide spectrum of engineering and scientific operations. For this reason, we explore four different approaches based on OmpSs, that aim to increase the performance attained by SpMV thanks to making better use of the platform resources. In this section, we present these approaches and provide a small schema and pseudo-code to illustrate each case.

3.1 One task per row

One task per row is a simple and straight-forward approach in which one task per row is created (see the pseudo-code and schema in Figure 1). Given that each task deals with a different row, there are no dependencies. However, numerous tasks are created, as many as rows are in the matrix; and, these tasks are usually very small due to the low amount of non-zeros per row, thus introducing a non-negligible overhead for the runtime. In addition, the workload unbalance is inherent to this approach since the number of computations performed by each task depends on the number of non-zero elements.

```
for ( r = 0; r < nRows; r++){
    sval = 0.0;
    #pragma oss task ...
    {
        for ( c = 0; c < nCols; c++) {
            val = VAL_A[ROW_A[r]+c];
            col = COL_A[ROW_A[r]+c];
            sval += val* X[col]*ALPHA;
        }
        Y[r] = sval + Y[r] * BETA;
    }
}
```

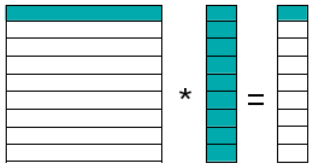

Fig. 1: Pseudo-code and schema for one task per row approach.

3.2 Blocking

Blocking implementation consists of splitting the matrix into smaller blocks and creating one task per block. With this strategy we ensure the reuse of the same entries of the array y within the task, thus improving data locality. Nevertheless, blocking the matrix requires a preprocessing in order to create the blocks in CSR format, which may add an overhead to the total run time. Moreover, all the blocks that comprise the same rows in the matrix update the same positions of the array y, turning into data dependencies. An additional question to take into account with this approach is the changes required in the code in order to apply blocking, since restructuring the matrix and dealing with the new data

dependencies make the programming difficult. Moreover, the block size to be used when blocking the matrix needs to be calculated in advance, requiring a previous analysis to determine it.

This approach is based on the code developed in [28], where an improved version of the conjugate gradient method is presented.

3.3 Taskloop

Keeping the use of coarser tasks, we propose the use of *taskloop*. In this case, each task will perform the matrix-vector multiplication on a fixed number of rows. The *taskloop* construct is used to distribute the rows in different tasks and the clause *grainsize* is used to determine the number of rows processed by each task. The main advantage of this approach is its simplicity, although the *grainsize* needs to be determined to maximize the use of the cores. However, it is important to note that the number of non-zeros may be highly unbalanced depending on the matrix. In our case, the *grainsize* is set in order to create one chunk per core, thus it is calculated as $\#rows/\#cores$. In this way, we ensure that all cores are used and the overhead due to tasks creation is minimum. Thus, it can be used as a baseline.

```
#pragma oss taskloop
            grainsize (nRows / #cores)
for ( r = 0; r < nRows; r++){
    sval = 0.0;
    for ( c = 0; c < nCols; c++) {
            val = VAL_A[ROW_A[r]+c];
            col = COL_A[ROW_A[r]+c];
            sval += val* X[col]*ALPHA;
     }
     Y[r] = sval + Y[r] * BETA;
}
```

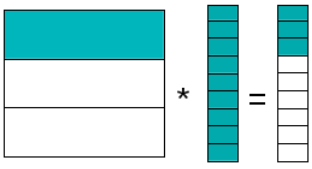

Fig. 2: Pseudo-code and schema for *taskloop* approach.

3.4 Grouping

Finally, aiming to keep using coarse tasks but trying to adapt to the different amount of non-zero elements per row, we propose to apply the *grouping* approach of Valero-Lara et al. [16, 22]. In this case, we create groups of rows according to a limit (given by the architecture, e.g. L1 size, L2 size, ...) and each group is processed by a different task. The main drawback of this approach is that it requires extra calculations in order to create the groups and this makes the code less readable. Also, using this approach one core is busy computing the next group and creating tasks.

4 Performance analysis

In this section, we present performance results for all the presented approaches in order to show the benefits/drawbacks of each one.

We have used a set of 12 characteristic matrices obtained from the SuiteSparse Matrix Collection [7] (formerly the University of Florida Sparse Matrix Collection)[6].

Although we analyze all the matrices of our test set, we pay particular attention at the *in-2004* matrix as the main test case, due to its characteristics. The *in-2004* matrix is a non-symmetric square matrix with 1,382,908 rows and 16,917,053 non-zero elements. Additionally, as reported in Table 1, it has rows with no elements (minimum 0) and other rows with quite a few elements (maximum 7753). These features made us consider this matrix as an "extreme" test case in which sparsity in unevenly present.

A graphical representation of the in-2004 example matrix is shown in the last column of Table 1.

Performance results

We have run our tests on Marenostrum 4 and Dibona clusters; we have used a single node of Marenostrum 4 Supercomputer, featuring two sockets Intel Xeon Platinum 8160 CPU with 24 cores each at 2.10GHz for a total of 48 cores per node. Regarding memory hierarchy, each core has 32KB L1 and 1MB L2 caches, and 33MB L3 cache shared among the 24 cores per socket. Regarding Dibona, each node presents two sockets ARM Thunder X2 (ARMv8 NEON) CPU with 32 cores each running at 2.0GHz for a total of 64 cores per node. In this case, only one socket has been considered for our tests. The memory hierarchy characteristics for this platform are 32KB of L1 cache, 256KB L2 cache, and 32MB L3 cache.

All tests are compiled with mcxx 2.3.0 (with GCC 6.4.0 or Intel icc 17.0.4 if available) and OmpSs-2 2018.06 (nanos6 2.4); for those tests that use MKL functions, MKL 2017.4 is used. Each test is run 20 times given the short time required for the computation on SpMV; from this measurements, the first repetition is discarded and only used as a warm-up phase. The reported values are calculated as the median of the remaining 19 repetitions, which measure exclusively the computation of the SpMV, leaving outside the initialization of the operands. Moreover, in each repetition cache memory is flashed to avoid data reuse between consecutive tests. In order to palliate possible NUMA effects on the overall execution time, affinity is set via *taskset* and $numactl --interleave = all$ is used to spread across the sockets.

Figure 3-Left graphically illustrates execution time for single-threaded MKL (*mkl_dcsrmv*), and the multi-threaded one (*mkl_sparse_d_mv*) as reference. The

[6] Input matrices from the UFMC are: cant, conf5_4-8x8-05, consph, cop20k_A, eu-2005, Ga41As41H72, in-2004, mac_econ_fwd500, mpi1, pdb1HYS, Si41Ge41H72, webbase1-M

single-threaded MKL routine implements SpMV as described in Equation 1 on a sparse matrix stored in CSR format, however, the multi-threaded MKL routine performs the same operation in parallel, but it requires the use of specific MKL structures to deal with the CSR matrix. Note that the order of the matrices in the x differs from 1, showing decreasing performance to ease the reading of the plots.

Table 1: Set of matrices used in SpMV tests. Information provided for each matrix: matrix ID, name in the SuiteSparse Matrix Collection, domain, number of rows (and columns), number of non-zero elements, maximum non-zeros per row, minimum non-zeros per row, average non-zeros per row, image of the matrix.

ID	Name	Domain	#rows	NNZ	Max.	Min.	Avg.	Matrix
m1	cant	FEM Cantilever	62,451	2,034,917	40	1	32	
m2	conf5_4-8x8-05	Quantum chromodynamics	49,152	1,916,928	39	39	39	
m3	consph	FEM concentric spheres	83,334	3,046,907	66	1	36	
m4	cop20k_A	Accelerator cavity design	121,192	1,362,087	24	0	11	
m5	eu-2005	Small web crawl of .eu domain	862,664	19,235,140	6,985	0	22	
m6	Ga41As41H72	Real-space pseudo potential method	268,296	9,378,286	472	1	34	
m7	**in-2004**	Small web crawl of .in domain	1,382,908	16,917,053	7,753	0	12	
m8	mac_econ_fwd500	Macroeconomic model	206,500	1,273,389	44	1	6	
m9	mip1	Optimiation problem	66,463	5,209,641	713	1	78	
m10	pdb1HYS	Protein data bank 1HYS	36,417	2,190,591	184	1	60	
m11	Si41Ge41H72	Real-space pseudo potential method	185,639	7,598,452	531	1	40	
m12	webbase1-M	Web connectivity matrix	1,000,005	3,105,536	4,700	1	3	

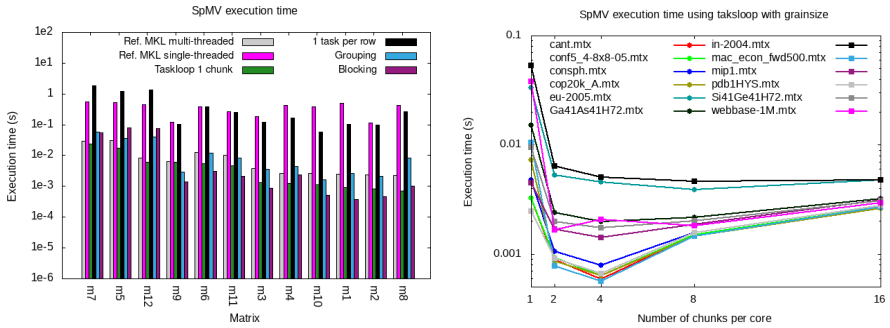

Fig. 3: Execution time for SpMV with different approaches on Intel Xeon: *one-task-per-row*, *grouping*, *blocking* and *taskloop* (Left). Execution time for 1, 2, 4, 8, and 16 chunks per core (Right).

Performance results show that the reference single-threaded MKL routine and the *one-task-per-row* approach, provide longer execution times.

For the *one-task-per-row* approach, this behavior was already predicted when presenting this strategy since many tasks are created (as many as rows) and its granularity is too fine, introducing a relevant overhead.

According to these results the best options to parallelize SpMV are *grouping*, *multi-threaded MKL*, *blocking* and *taskloop* strategies. *Grouping* (with a limit equal to 25% of L2 cache, being this the best limit tested) provides the worst performance among these three options. *MKL multi-threaded* presents a behavior similar to *Grouping*, although it performs better for very unbalanced matrices (m4, m5, m7, and m12), being slower than *taskloop* on all the tested matrices.

Blocking seems the best option in terms of execution time in some of the cases. However, execution time is considerably high for those matrices that have a highly unbalanced number of non-zeros per row (m4, m5, m7, and m12) and, more important, the preprocessing time needed to block the input matrix as CSR subblocks makes it unfeasible since this preprocessing requires an execution time between 2 and 3 orders of magnitude greater than the SpMV execution time. Finally, *taskloop* provides good results in all cases and, besides, eases the parallelization of SpMV thanks to its simplicity, facilitating the maintainability of the code.

In the light of the presented performance results, we consider the *taskloop* approach the most suitable one in order to parallelize SpMV. This selection is

Fig. 4: Traces for SpMV when applying *taskloop* with 1 (first) and 4 chunks (second) per core, *taskloop + nesting* with $th = 25\%\ L2$ (third) and $th = avg\ nnz$ per chunk (fourth) optimizations to taskloop approach.

based on several reasons such as i) the fact that it is the easiest approach since it only requires using the *taskloop* construct, ii) it is also easily optimizable because, although it requires a previous analysis, testing different grain sizes on the platform is enough to attain a reasonable behavior, iii) it follows the OpenMP standard, so portability is ensured even if OmpSs is not available on other platforms.

Figure 4 (first) contains the trace of the execution of SpMV (using *in-2004* as an input) based on the taskloop strategy when a grainsize of $\#rows/\#cores$ is used. Axis y shows the 48 cores executing the kernel (on Intel Xeon platform) and axis x is time. The trace shows that the taskloop construct maximizes the use of resources, using all the available cores in the platform. However, due to the static partitioning of the iterations made by taskloop and the unbalanced nature of the created chunks, the total execution time for a few tasks is well above the average task execution time ($\sim 24,000\mu$s vs. $8,000\mu$s). In this scenario, and given the good use of the resources made by the taskloop construct, we consider exploring other alternatives that could potentially palliate the imbalance among tasks and thus reduce the overall execution time.

5 Optimizing the *taskloop* implementation

In this section, we present two approaches to improve the load balance of SpMV when using the *taskloop* construct to distribute the computations among the cores. First, we focus on the straight-forward use of the *taskloop* construct and the *g*rainsize clause, performing an analysis in order to find the most suitable *grainsize*. As an alternative to this approach, we present a more sophisticated strategy where two levels of parallelism are created depending on a few features either of the architecture or the input matrix.

5.1 Taskloop grainsize selection

As mentioned before, although the *taskloop* strategy provides high performance, it is essential to determine the *grainsize* used to create the tasks. To find this number, and keeping in mind that we want to maximize the use of the resources, we tested different configurations that distribute the number of rows evenly (independently of the number of non-zero elements in each row). In addition, it is necessary to create enough tasks to "feed" all the available cores, for this reason, we analyzed the performance of the SpMV when creating $\#cores * factor$ tasks, with *factor* equal to 1, 2, 4, 8 and 16, and $\#cores$ equal to 48. This formula computes the size of the *grainsize* of the *taskloop* clause and then the number of tasks as well. Figure 3-Right graphically illustrates the execution time for all the matrices of the test set (Table 1) using different factors. We can see that, even the matrices are very different among them in terms of number of non-zero elements and sparsity, results show that almost all matrices present the same behavior, finding the minimum execution time when a factor of 4 is used. Note that this is not the case for *eu-2005* and *webbase-1M* matrices. For these matrices a factor of 8 provides lower execution times.

5.2 Taskloop + nesting

Finally, we present *taskloop + nesting* as an alternative to create tasks with a more regular number of non-zero elements, thus trying to mimic the behavior of *grouping* but reducing the overhead introduced by the thread in charge of creating the groups.

In this scenario, first we need to replace the *taskloop* construct used to create the chunks by a *task* construct. This change allows us to know the first and last row that is processed in a specific chunk and, consequently, the number of non-zeros of the chunk can easily be calculated. Despite this change, we set the number of rows to be processed by a task to $\#rows/\#cores$, which mimics the behavior of setting the *grainsize* clause for the first level *taskloop* to the same number. Then, a second level of parallelism is created in order to balance the workload among the created tasks when necessary. The idea is subdividing those tasks created at the first level that have a huge number of non-zeros into smaller tasks that can be balanced better. To this end, every time a task is created at the first level we check if the number of non-zeros of the chunk being processed is greater than a threshold *th*. This idea is presented in Figure 5.

```
nChunks = get_num_chunks(nRows);
for ( nc = 0; nc < nChunks; nc++){
  #pragma oss task
  {
    nnzT = number_of_non_zeros_in_chunk(nc);
    init_row = get_init_row(nc);
    end_row = get_end_row(nc);
    #pragma oss taskloop
                num_tasks(nnzT/th)
                if (nnzT > th)
    for(r = init_row; r < end_row; r++){
          sval = 0.0;
          for ( c = 0; c < nCols; c++){
               val = VAL_A[ROW_A[r]+c];
               col = COL_A[ROW_A[r]+c];
               sval += val * X[col] * ALPHA;
          }
          Y[r] = sval + Y[r] * BETA;
    }
  }// End of pragma
}
```

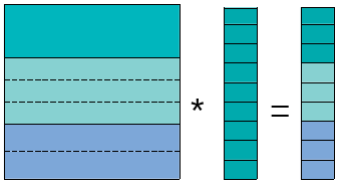

Fig. 5: Pseudo-code and schema for *taskloop + nesting* approach.

To set the threshold value we have followed two different strategies, one focused on the architecture features and one that takes into account the sparsity of the matrix. In the first case, we set the threshold to a specific value that depends on the L2 cache size, more specifically, we perform the tests setting the threshold to 25% and 50% of L2 capacity. For the second case, we calculate the average number of non-zeros per chunk, this is the total amount of non-zero elements in the matrix divided by the number of cores of the platform. In both cases, if the number of non-zero elements of the chunk is greater than the

threshold *th*, the task is split in as many tasks as necessary, each of them in charge of *th* elements.

To make a deeper analysis, Figure 4 shows the traces for the following strategies on *in-2004* matrix: *taskloop* with 4 chunks per core, *taskloop + nesting* with *th*=25% of L2 cache, and *taskloop + nesting* with *th*=average of nnz per chunk. Axis *y* shows the 48 cores running SpMV kernel on Intel Xeon platform, while axis *x* shows the execution time. All traces are in the same scale; this is, the total time represented by axis *x* is the same in all cases.

After the analysis of the traces for *in-2004* matrix, we can state that applying nesting may be beneficial in order to compact the trace by splitting the most time consuming tasks in smaller ones. In this specific case, the approach focused on architecture features, setting the threshold to 25% of L2 cache, allows to compact the trace by creating smaller tasks, which are scheduled in a more balanced way and, consequently, help to reduce the overall execution time. However, setting the threshold to the average number of non-zeros per row, generates similar imbalance to that seen in *taskloop*.

We extend the analysis to all the matrices of the test set (see Figure 6). We use the performance of the *taskloop one chunk per core* approach as a reference. For well structured matrices, where the number of non-zero elements per row remains almost constant, the *taskloop 4 chunks per core* approach is able to achieve good performance; almost negligible overhead is introduced and workload is well distributed thanks to the nature of the matrices (m1, m2, and m3). However, for very unbalanced matrices (m5, m7, m9, and m12), using *taskloop + nesting* based on L2 capacity is able to outperform the previous approach, achieving about 60% faster executions with respect to the reference parallel implementation. Regarding the *taskloop + nesting* approach based on the average number of non-zeros per chunk, we see that performance is similar to that attained in the other nested approaches except for a few matrices (m6, m9, m11), where the execution time is considerably increased. Figure 6 also includes the percentage of improvement for *taskloop 4 chunks per core* and *taskloop + nesting* based on L2 capacity (higher is better) with respect to the parallel reference code. Results show that the gains when cache capacity is taken into account are relevant especially for very unbalanced matrices; however, for balanced matrices the *taskloop 4 chunks per core* approach provides better results.

When comparing these results with those obtained for ThunderX2 (Figure 7), we observe a similar behavior. The only exceptions are m6, m9 and m11 matrices. For those matrices, slightly higher performance is attained with taskloop with 4 chunks when Intel Xeon is used.

6 Conclusions and future work

Performance results show that making a static and homogeneous partition of the rows by using *taskloop* is able to achieve a good result on well-balanced sparse matrices. However, on other matrices where we find an important unbalanced sparsity, the use of *taskloop + nesting* presents a much better behavior, achieving

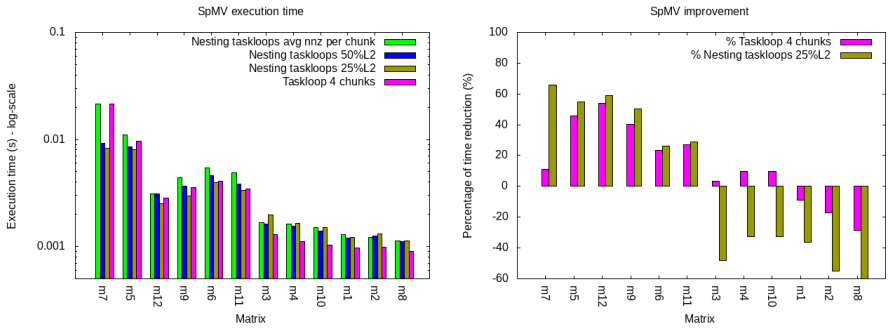

Fig. 6: Execution time (left) and percentage gain (right) for SpMV when applying optimizations to *taskloop* on Intel Xeon platform.

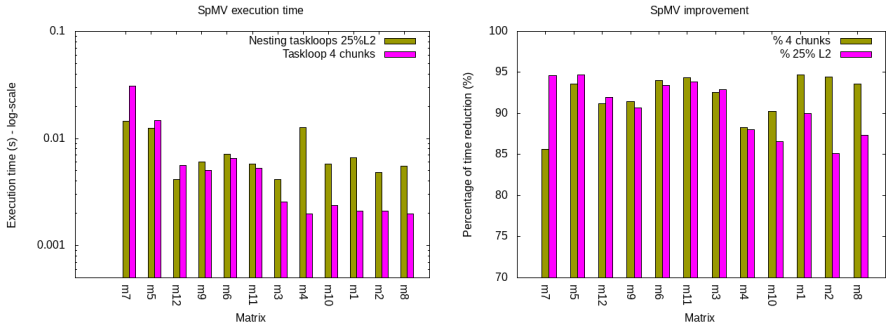

Fig. 7: Execution time (left) and percentage gain (right) for SpMV when applying optimizations to *taskloop* on ThunderX2 platform.

an important time reduction in some cases. Both approaches are faster than the multi-threaded MKL counterpart.

In this scenario, we plan as future work to combine both strategies via the *final* clause in other to choose the most appropriate one, depending on the input matrix with the aim of attaining higher performance in each case.

References

1. cuSparse. https://docs.nvidia.com/pdf/CUSPARSE_Library.pdf
2. OmpSs-2. https://pm.bsc.es/ftp/ompss-2/doc/spec/OmpSs-2-Specification.pdf
3. Alnæs, M.S., Blechta, J., Hake, J., Johansson, A., Kehlet, B., Logg, A., Richardson, C., Ring, J., Rognes, M.E., Wells, G.N.: The FEniCS Project Version 1.5. Archive of Numerical Software **3**(100) (2015). https://doi.org/10.11588/ans.2015.100.20553
4. Amestoy, P.R., Duff, I.S., L'excellent, J.Y.: Multifrontal parallel distributed symmetric and unsymmetric solvers. Computer methods in applied mechanics and engineering **184**(2-4), 501–520 (2000)

5. Anzt, H., Sawyer, W., Tomov, S., Luszczek, P., Yamazaki, I., Dongarra, J.: Optimizing Krylov Subspace Solvers on Graphics Processing Units. In: Fourth International Workshop on Accelerators and Hybrid Exascale Systems (AsHES), IPDPS 2014. IEEE, IEEE, Phoenix, AZ (05-2014 2014)
6. Balay, S., Abhyankar, S., Adams, M.F., Brown, J., Brune, P., Buschelman, K., Dalcin, L., Dener, A., Eijkhout, V., Gropp, W.D., Kaushik, D., Knepley, M.G., May, D.A., McInnes, L.C., Mills, R.T., Munson, T., Rupp, K., Sanan, P., Smith, B.F., Zampini, S., Zhang, H., Zhang, H.: PETSc Web page. http://www.mcs.anl.gov/petsc (2018), http://www.mcs.anl.gov/petsc
7. Davis, T.A., Hu, Y.: The University of Florida Sparse Matrix Collection. ACM Trans. Math. Softw. **38**(1), 1:1–1:25 (Dec 2011). https://doi.org/10.1145/2049662.2049663, http://doi.acm.org/10.1145/2049662.2049663
8. Dongarra, J.J., Hammarling, S., Higham, N.J., Relton, S.D., Valero-Lara, P., Zounon, M.: 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. pp. 495–504 (2017). https://doi.org/10.1016/j.procs.2017.05.138
9. Dongarra, J.J., Heroux, M.A., Luszczek, P.: HPCG Benchmark : a New Metric for Ranking High Performance Computing Systems (2015)
10. Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures. Parallel Processing Letters **21**(2), 173–193 (2011). https://doi.org/10.1142/S0129626411000151, https://doi.org/10.1142/S0129626411000151
11. Langr, D., Tvrdík, P.: Evaluation Criteria for Sparse Matrix Storage Formats. IEEE Transactions on Parallel and Distributed Systems **27**(2), 428–440 (Feb 2016). https://doi.org/10.1109/TPDS.2015.2401575
12. Li, X.S.: An Overview of SuperLU: Algorithms, Implementation, and User Interface. ACM Trans. Math. Software **31**(3), 302–325 (September 2005)
13. Llort, G., Servat, H., Gonzalez, J., Giménez, J., Labarta, J.: On the usefulness of object tracking techniques in performance analysis. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013. pp. 29:1–29:11 (2013)
14. Ohshima, S., Katagiri, T., Matsumoto, M.: Performance Optimization of SpMV Using CRS Format by Considering OpenMP Scheduling on CPUs and MIC. In: Proceedings of the 2014 IEEE 8th International Symposium on Embedded Multicore/Manycore SoCs. pp. 253–260. MCSOC '14, IEEE Computer Society, Washington, DC, USA (2014). https://doi.org/10.1109/MCSoC.2014.43, http://dx.doi.org/10.1109/MCSoC.2014.43
15. Ortega, G., Vázquez, F., García, I., Garzón, E.M.: FastSpMM: An Efficient Library for Sparse Matrix Matrix Product on GPUs. The Computer Journal **57**(7), 968–979 (2014). https://doi.org/10.1093/comjnl/bxt038, http://dx.doi.org/10.1093/comjnl/bxt038
16. Valero-Lara, P., Martinez-Perez, I., Mateo, S., Sirvent, R., Beltran, V., Martorell, X., Labarta, J.: Variable Batched DGEMM. In: 2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP). vol. 00, pp. 363–367 (Mar 2018). https://doi.org/10.1109/PDP2018.2018.00065, doi.ieeecomputersociety.org/10.1109/PDP2018.2018.00065

17. Valero-Lara, P., Andrade, D., Sirvent, R., Labarta, J., Fraguela, B.B., Doallo, R.: A Fast Solver for Large Tridiagonal Systems on Multi-Core Processors (Lass Library). IEEE Access **7**, 23365–23378 (2019). https://doi.org/10.1109/ACCESS.2019.2900122
18. Valero-Lara, P., Catalán, S., Martorell, X., Labarta, J.: BLAS-3 Optimized by OmpSs Regions (LASs Library). In: 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2019, Pavia, Italy, February 13-15, 2019. pp. 25–32 (2019). https://doi.org/10.1109/EMPDP.2019.8671545
19. Valero-Lara, P., Catalán, S., Martorell, X., Usui, T., Labarta, J.: sLASs: A fully automatic auto-tuned linear algebra library based on OpenMP extensions implemented in OmpSs (LASs Library). Journal of Parallel and Distributed Computing **138**, 153 – 171 (2020). https://doi.org/https://doi.org/10.1016/j.jpdc.2019.12.002
20. Valero-Lara, P., Martínez-Pérez, I., Peña, A.J., Martorell, X., Sirvent, R., Labarta, J.: cuHinesBatch: Solving Multiple Hines systems on GPUs Human Brain Project*. In: International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland. pp. 566–575 (2017). https://doi.org/10.1016/j.procs.2017.05.145
21. Valero-Lara, P., Martínez-Pérez, I., Sirvent, R., Martorell, X., Peña, A.J.: cuThomasBatch and cuThomasVBatch, CUDA Routines to compute batch of tridiagonal systems on NVIDIA GPUs. Concurrency and Computation: Practice and Experience **30**(24) (2018). https://doi.org/10.1002/cpe.4909
22. Valero-Lara, P., Sirvent, R., Peña, A.J., Martorell, X., Labarta, J.: MPI+OpenMP Tasking Scalability for the Simulation of the Human Brain: Human Brain Project. In: Proceedings of the 25th European MPI Users' Group Meeting. pp. 5:1–5:8. EuroMPI'18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3236367.3236373, http://doi.acm.org/10.1145/3236367.3236373
23. Valero-Lara, P., Sirvent, R., Peña, A.J., Labarta, J.: MPI+OpenMP tasking scalability for multi-morphology simulations of the human brain. Parallel Computing **84**, 50–61 (2019). https://doi.org/10.1016/j.parco.2019.03.006, https://doi.org/10.1016/j.parco.2019.03.006
24. Valero-Lara, P., Sirvent, R., Peña, A.J., Martorell, X., Labarta, J.: 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. pp. 5:1–5:8 (2018). https://doi.org/10.1145/3236367.3236373
25. Valero-Lara, Pedro, Martínez-Pérez, Ivan, Sirvent, Raül, Peña, Antonio J., Martorell, Xavier, Labarta, Jesús: Simulating the behavior of the Human Brain on GPUs. Oil Gas Sci. Technol. - Rev. IFP Energies nouvelles **73**, 63 (2018). https://doi.org/10.2516/ogst/2018061, https://doi.org/10.2516/ogst/2018061
26. Ye, F., Calvin, C., Petiton, S.G.: A Study of SpMV Implementation Using MPI and OpenMP on Intel Many-Core Architecture. In: VECPAR (2014)
27. Zhang, Y., Li, S., Yan, S., Zhou, H.: A Cross-Platform SpMV Framework on Many-Core Architectures. ACM Trans. Archit. Code Optim. **13**(4), 33:1–33:25 (Oct 2016). https://doi.org/10.1145/2994148, http://doi.acm.org/10.1145/2994148
28. Zhuang, S., Casas, M.: Iteration-fusing Conjugate Gradient. In: Proceedings of the International Conference on Supercomputing. pp. 21:1–21:10. ICS '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3079079.3079091, http://doi.acm.org/10.1145/3079079.3079091