© 2018 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes,creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.
DOI 10.1109/PDP2018.2018.00065

Variable Batched DGEMM

Pedro Valero-Lara, Ivan Martínez-Pérez, Sergi Mateo, Raül Sirvent, Vicenç Beltran
Barcelona Supercomuting Center (BSC)
Barcelona, Spain
{*pedro.valero, ivan.martinez, sergi.mateo, raul.sirvent, vbeltran*}*@bsc.es*

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

***Abstract*—Many scientific applications are in need to solve a high number of small-size independent problems. However, these individual problems do not provide enough parallelism to efficiently exploit the current parallel architectures, and then, these must be computed as a *batch* in order to saturate the hardware. Today, vendors such as Intel and NVIDIA are developing their own suite of *batch* routines. Although most of the works focus on batches of fixed size, in real applications we can not assume a uniform size for all set of problems, so it is necessary the study of new efficient approaches to deal with batches of variable size, which is more challenging compared to batches of fixed size. We explore and analyze different strategies based on *parallel for, task and taskloop OpenMP pragmas*. These strategies are straightforward from programmer's point of view, however they present a very different impact on performance. We also analyze a new prototype provided by Intel (*MKL*), which deals with *batch* operations (*cblas_dgemm_batch*). Basically, these techniques attempt to distribute subgroups composed by the same number of problems without considering the differences among them. For this reason that we propose a new approach called *grouping*. It basically groups a set of problems until filling a limit in terms of memory occupancy or number of operations. In this way, groups composed by different number of problems are distributed on cores, achieving a more balanced distribution in terms of computational cost. However, these strategies must be tuned to achieve a good performance, and so we also explore and evaluate some off-line auto-tuning strategies on the proposed schedulers.**

***Keywords*-Batched BLAS; Off-line Auto-tunning; Runtime; OpenMP; DGEMM; Intel Xeon; Intel Xeon KNL;**

I. Introduction

Although it is possible to find a high number of different HPC libraries that face the parallel implementation of Linear Algebra problems, such as PLASMA [1], Intel MKL [2], NVIDIA cuBLAS [3], among others [1], we find a lack of efficient implementations for some of the most critical applications currently of high interests in the HPC community. For instance, tensor contractions [2], [3], [4] for deep learning and low rank matrix computations [5] are key operations. These applications are in need of computing thousands of independent dense linear algebra operations (*batch*) on small matrices [6], [7]. It is possible to find some works that address this problem, that is, computing a very high number of independent Basic Linear Algebra (BLAS) routines. Most of these works are focused on batches of fixed size (*batch fixed*), using GPUs [8], [4], [9], [10], [11], where the problems to be computed share the same size. Recently, Ahmad Abdelfattah et al. [12] presented a set of heterogeneous CPU+GPU strategies to deal with batches of variable size (*batch variable*), where they use *OpenMP parallel for dynamic* pragmas to deal with *batch variable* on multicore CPU. As shown in this paper, there are also other interesting approaches that can outperform the performance achieved by *parallel for dynamic*, in particular for small matrices. In this paper, the authors analyze a set of strategies, such as other *parallel for* schedulers, *tasks*, *priorities* and *taskloop*, among others. Also, we propose a new strategy called *grouping* to deal with *batch variable*, which are able to distribute no homogeneous bins of *DGEMMs* on cores, achieving a more balanced computational load. However some approaches, in particular those based on *taskloop* and *grouping*, must be tuned to guarantee a good performance, and so we also study *auto-tuning* strategies on these last approaches.

For our experiments we have used two different architectures: Intel Xeon and the self-hosted Intel Xeon KNL. We evaluate how the different characteristics of these two platforms influence on performance, when dealing with batches of variable size. We pay a particular attention to the memory hierarchy since, as it shown in the work, this has important consequences on performance.

This paper is structured as follows. Section II briefly introduces the strategies selected to cope with *batch variable*, paying particular attention to the proposed strategy, *grouping*. In Section III we deeply analyzes the pros and cons of each of the strategies tested. This Section also includes the study to evaluate the off-line auto-tuning strategies on *taskloop* and *grouping*. Finally, the conclusions and future road map are outlined in Section IV.

[1] https://bitbucket.org/icl/plasma
[2] https://software.intel.com/en-us/intel-mkl
[3] http://docs.nvidia.com/cuda/cublas

II. BATCH VARIABLE

We explore different approaches that manage in a different way, how the set of *DGEMMs* are distributed. Table I summarizes the different approaches evaluated.

All of them (except the prototype *cblas_dgemm_batch* of *MKL* [4], which we do not know how is implemented) compute sequential calls of *MKL cblas_dgemm*. One of the approaches evaluated is the one based on *parallel for static*. This implementation makes use of a simple scheduler, but it is in need of dealing with a potential unbalancing due to the different size of our matrices. For this reason we explore other *parallel for schedulers*. Next we describe the main characteristics of each of the schedulers considered in the present work:

- *static*: when `schedule(static, chunk_size)` is specified, iterations are divided into chunks of size chunk_size, and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number. When no chunk_size is specified, the iteration space is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each thread. The size of the chunks is unspecified in this case.
- *dynamic*: when `schedule(dynamic,chunk_size)` is specified, the iterations are distributed to threads in the team in chunks. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed. When no chunk_size is specified, it defaults to 1.
- *guided*: when `schedule(guided, chunk_size)` is specified, the iterations are assigned to threads in the team in chunks. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be assigned. For a chunk_size of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads in the team, decreasing to 1. For a chunk_size with value k (greater than 1), the size of each chunk is determined in the same way, with the restriction that the chunks do not contain fewer than k iterations (except for the chunk that contains the sequentially last iteration, which may have fewer than k iterations). When no chunk_size is specified, it defaults to 1.
- *auto*: when `schedule(auto)` is specified, the decision regarding scheduling is delegated to the compiler and/or runtime system. The programmer gives the implementation the freedom to choose any possible mapping of iterations to threads in the team.
- *runtime*: when `schedule(runtime)` is specified, the decision regarding scheduling is deferred until run time, and the schedule and chunk size are taken from the run-sched-var ICV. If the ICV is set to auto, the schedule is implementation defined.

The implementations based on *dynamic*, *guided*, *auto* and *runtime* increase the overhead caused by the scheduler but can reduce the unbalancing found in *batch variable*.

Other interesting approaches are those based on tasks, in which we launch one task per *DGEMM*. Similar to the previous approach, we also evaluate the use of priorities coupled with tasks (a feature introduced in *OpenMP* 4.5). To use this approach, first we must set the maximum priority via the environment variable *OMP_MAX_TASK_PRIORITY*. This variable is set as follow: ($M_{max} \times K_{max} \times N_{max}$), being M_{max}, K_{max} and N_{max} the maximum size used for each dimension. Then we assign one priority to every *DGEMM* (task) according to its size ($M_i \times K_i \times N_i$). The motivation of this is to compute the bigger problems before the smaller ones, in order to achieve a better balancing.

We also explore the use of *taskloop*. Using this strategy we can set the number of iterations (number of *DGEMMs*) per core using the *grainsize* clause. However, this approach is still too rigid and then, we have the same problem than in the previous approaches, that is, uniform number of *DGEMMs* are distributed on cores.

A. Grouping

Next we describe the implementation of *grouping*. Basically, it is a master-slave model, where one of the tasks (master) go through the *batch*, grouping *DGEMMs* until filling a limit. Once the master has grouped a set of *DGEMMs*, this creates a new task (slave), which computes the set of *DGEMMs* grouped by the master. All the slaves are launched asynchronously. In this way, while the master is grouping a set of *DGEMMs* and launching tasks, these lasts compute the *DGEMMs* assigned by the master. We propose two different heuristics to compute the limit, one in terms of memory occupancy and one in terms of number of operations. In the case of memory occupancy, we group according to the accumulative size[5]:

$$sizeof(M_i \times K_i) + sizeof(K_i \times N_i) + sizeof(M_i \times N_i) < limit, \forall i = 0, \ldots, BATCH_COUNT - 1 \quad (1)$$

On the other hand, in the case of number of operations, we group according to the accumulative computational cost:

$$Op(M_i, K_i) + Op(K_i, N_i) + Op(M_i, N_i) < limit_{\#Op} \quad (2)$$

Unlike the previous approaches, *grouping* is more flexible and allows us to distribute non-uniform groups of *DGEMMs* on cores.

[4] https://software.intel.com/en-us/articles/introducing-batch-gemm-operations

[5] $A_i = M_i \times K_i, B_i = K_i \times N_i, C_i = M_i \times N_i, \forall i = 0, \ldots, \#DGEMMs$

Table I
LIST OF THE DIFFERENT APPROACHES EVALUATED

Approach	Description
MKL Batch DGEMM	cblas_dgemm_batch
OpenMP parallel for static	#pragma omp parallel for
OpenMP parallel for dynamic	#pragma omp parallel for schedule(dynamic)
OpenMP parallel for guided	#pragma omp parallel for schedule(guided)
OpenMP parallel for auto	#pragma omp parallel for schedule(auto)
OpenMP parallel for auto	#pragma omp parallel for schedule(runtime)
OpenMP taskloop	#pragma omp taskloop grainsize(GRAINSIZE)
OpenMP task	#pragma omp task
OpenMP task+priority	#pragma omp task priority(PRIORITY)
Grouping	Based on #pragma omp task

III. PERFORMANCE ANALYSIS

Our test case consists of computing a *batch* of 10,000 *DGEMMs*. The machines used in this experiment is a NUMA node with 2 sockets, using Xeon CPU E5-2630 v3 (Xeon), see Table II for more details, and an Xeon Phi 7230 “Knight Landing” (KNL), see Table III for more details. Hyperthreading is not enabled.

Table II
DETAILS OF THE ARCHITECTURE USED

Platform	Xeon E5-2630 v3 (Haswell) at 2.4 GHz
Cores	2×8
On-chip Memory	L1 256KB (per core)
	L2 2MB (unified)
	L3 20MB (unified)
Main Memory	128GB DDR4
Compiler	gcc 6.2.0
MKL	2017.1

All the matrices are chosen to have elements taken from a random uniform distribution. We set the seed[6] in order to execute the same cases (random distribution) on the different approaches that we want to evaluate. Regarding the range among min. and max. size of the matrices computed in the batch, there is no a characteristic size, as it depends on the nature of the applications, and because of that we have considered different ranges (1:8, 8:16, 16:32 and 32:64) to cover a wide range of possible scenarios. Similar to matrix sizes, there is no a characteristic way in which how the data is mapped on a *NUMA* architecture, as it also depends on the nature of the applications. In this sense, we explore two possible scenarios, one where the data is distributed by interleaving the data between the nodes[7], and one where the data is initialized sequentially without interleaving the data among the nodes. Using `numactl`, we can distribute (interleave) the data among the different sockets of a NUMA node homogeneously. For example, if our NUMA platform has two sockets, the half of the data used by our code is stored in one of the two sockets and the rest in the other one. However, if we do not use `numactl`, most of the data (even all the data depending of the memory capacity of the socket and data required by the code to be executed) is stored in one socket. Note that throughout all of our experiments we ensure that the cache of each processor is flushed before every invocation of a batch operation, to avoid obtaining misleading performance results. This is consistent with observations by Whaley et al. [13]. The results are shown in terms of GFLOPS. The platform based on KNL as been set in *cache* mode (default mode). It means that the MCDRAM (Table III) is used as a L3 unified cache memory shared by all cores [14]. Although KNL can be configured in different modes, previous studies [15] have proven that there are not important differences in terms of performance using the different possible configurations for batch operations. We have used the default case (no chunck_size is specified) for those tests based on *parallel for OpenMP pragmas*, as the different scenarios regarding the size of the chunk are explored using *taskloop* + *grainsize* (see Subsection III-C).

[6]“`srand(1)`”

[7]“`numactl --interleave=all`”

Table III
DETAILS OF THE ARCHITECTURE USED

Platform	Xeon Phi 7230 (Knight Landing) at 1.3 GHz
Cores	64
On-chip Memory	L1 32KB (per core)
	L2 1MB (unified per 2 cores-tile)
	MCDRAM/L3 16GB (unified)
Main Memory	96GB DDR4
Compiler	gcc 6.3.0
MKL	2017.3

A. Parallel for & MKL Batch

First of all, we analyze the performance achieved by those approaches based on *OpenMP parallel for* pragmas on Xeon. Fig. 1 shows that there is no a big difference among the set of schedulers tested when interleaving the data between *NUMA* nodes, being slightly better the *guided* scheduler. However, when data is non-interleaved, the difference between schedulers is much bigger, being the *dynamic* scheduler much faster than the other schedulers. Also, when small matrices are computed, a much better performance

is achieved when data is non-interleaved. Using KNL the influence of `numactl` is not as clear as on Xeon (see Fig. 2) being the fastest the *guided* and the *auto* schedulers.

We have also evaluated the *MKL* prototype (*cblas_dgemm_batch*) for *batch variable*. This *MKL* routine is not able to outperform the approaches based on *OpenMP pragmas*. Recently, other works have evaluated this prototype for *batch fixed* [6], [16]. For *batch fixed*, the use of the *MKL* prototype is proven to be as fast as the use of *OpenMP parallel for static* [6], [16]. However, for *batch variable*, in particular when data is non-interleaved, the *MKL* routine suffers from an important fall in performance. The *cblas_dgemm_batch* shows better performance on KNL (Fig. 2), but it is still not faster than some *parallel for schedulers* when data is non-interleaved. Also, it is important to highlight that the performance achieved by both, the *MKL* routine and the *parallel for schedulers*, on KNL is very low when small matrices are computed. This is studied more in detail in the work presented by Dongarra et al. [10].

B. Task

The use of one *task* per *DGEMM* (Fig. 1) generates a big unbalancing due to the management (instantiation) of the tasks, which makes this approach not efficient for *batch variable*. This approach is not able to provide enough work per core, so that some cores are idle along the execution. The use of *tasks+priorities* does not suppose an improvement with respect to use of only *tasks*. Although, big *DGEMMs* are executed before small *DGEMMs*, we still have the same problem found in the previous approach. Also, the *OpenMP throttle* is equal to 24, which minimizes the impact to use this approach on bigger group of tasks. The locality can be not efficiently exploited for those *batches* with a small distance between big and small *DGEMMs*. All this makes *tasks+priorities* slower than *task* for the scenarios tested. As graphically illustrated in Fig. 2 the overhead of the instantiation of a such high number of tasks is bigger on KNL.

C. Taskloop

Using *taskloop*, the programmer can define the number of iterations to be computed by each core. Unlike the approaches based on *tasks* and *tasks+priorities*, here there is no overhead for scheduling, as the distribution of *DGEMMs* (number of iterations per core) is defined by the programmer at compilation time. Figure 3 graphically illustrates the GFLOPS achieved increasing the number of iterations (*grainsize*) to be computed by core. The performance can change considerably with small modifications of *grainsize* on Xeon. However, using KNL the fluctuation in terms of performance is not so high as in Xeon. On Xeon, a lower performance is achieved when the data is non-interleaved. This is because uniform set of *DGEMMs* are distributed on the set of cores. This forces a non uniform distribution in terms of computational cost, as the size of the matrices is different.

D. Grouping

Given the results obtained by the approaches based on *tasks+priorities* and *tasks*, where the performance is conditioned by the instantiation of such a high number of tasks (*DGEMMs*), we propose a different approach based on *grouping*. Using this approach, it is not necessary to create one task per *DGEMM*, but one task computes a set of them, which can minimize the overhead caused by the instantiation, and then provide a much better performance with respect to *task* and *tasks+priorities*. In Fig. 4 and 5 (TotalFlops is the total number of operations to compute the whole *batch*), two different approaches for *grouping* are tested, one based on memory occupancy (*grouping mem* in Fig. 4 and 5) and one based on number of operations (*grouping op* in Fig. 4 and 5).

Using *grouping-mem* we see a similar trend on both platforms, Xeon and KNL, independently of the size of the matrices and how the data is mapped on memory. In particular, the maximum performance is achieved in a similar range, in terms of memory occupancy (Fig. 4 and 5). However, in *grouping-op* the trend is different. Unlike *grouping-mem*, in *grouping-op* the peak is achieved in different positions regarding the number of operations. On Xeon, how the data is mapped on memory has important consequences on performance. In particular the larger are the matrices, the more number of operations are necessary, when the data is interleaved. However, we see a different behavior when data in non-interleaved between *NUMA* nodes. In this case, the best *limit* is located in between of the min and max *limit* tested. This makes difficult the tuning of this approach, as the programmer must be aware of how the data is stored in memory in order to identify the best *limit*. Unlike Xeon, using KNL we do not identify an important difference among the two kind of data storage, using and not using `numactl`. This makes much easier the use of auto-tuning strategies on the *grouping-op* approach when this architecture is used. Unlike *taskloop*, these approaches are in need of a task (master) to instantiate the other tasks, however it achieves a similar, even higher, performance than *taskloop*. Also, it is important to highlight that, once the maximum performance is achieved, the performance fluctuation is not as big as using *taskloop* on Xeon.

E. Off-line Auto-tuning

In this sub-section is explored the effectiveness of *off-line auto-tuning* strategies to deal with *batch variable*. *Off-line auto-tuning* are those strategies that attempt to tune the parameters, which are susceptible to be tunable, to adapt one specific problem to one specific architecture. The parameters

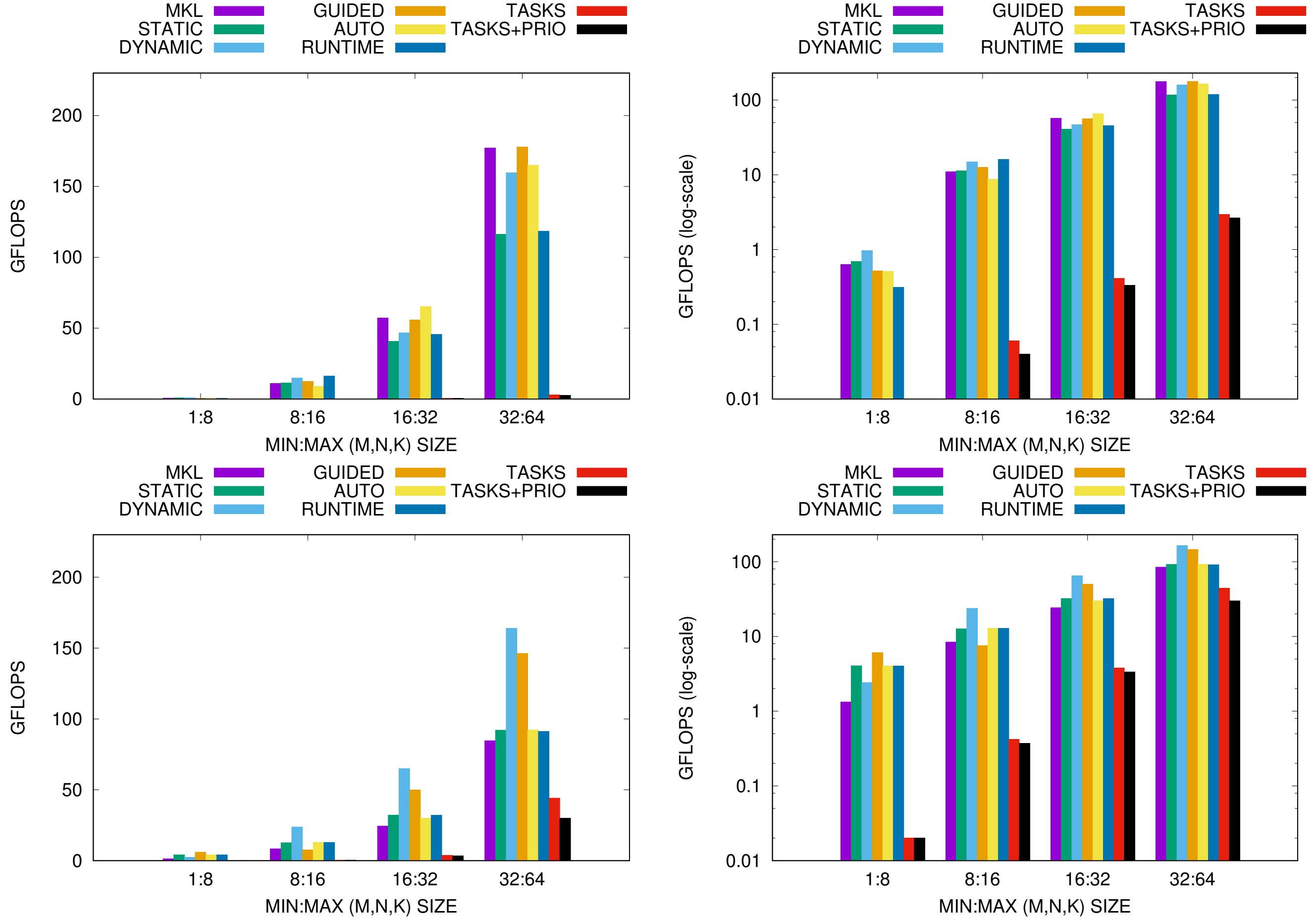

Figure 1. GFLOPS achieved by *MKL*, *parallel for*, *task* and *task+prio* to compute 10,000 *DGEMMs* for different size ranges (MIN:MAX): (1:8), (8:16), (16:32) and (32:64), interleaving (top) and not interleaving (bottom) the data between *NUMA* nodes.

are tuned before computing and they do not change along the execution.

First, we identify those tunable parameters for the approaches *grouping-mem*, *grouping-op* and *taskloop*. The parameter to be tuned for *taskloop* is the *grainsize*. In *grouping-op* and in *grouping-mem*, we have to choose the *limit* in terms of number of operations and the memory occupancy per core, respectively. In all these approaches, it is difficult to identify the best tuning a priori (see Fig. 3, 4 and 5). We set the *grainsize* parameter equal to $\#DGEMMs/\#cores$ for *taskloop*. For *grouping-mem*, we have chosen as the *limit*, the size of L1-cache (see Table II). As shown in Fig. 4, a peak of performance is achieved when the *limit* is located about the L1 capacity for Xeon. In KNL we see some differences with respect to the performance achieved by the Xeon (Fig. 5). This is because of the memory hierarchy of KNL. In KNL the L2-cache level is divided into 32 independent caches [14]. Each independent L2-cache is shared by two cores (this is known as tile). As graphically illustrated in Fig. 5, the peak in performance is not obtained when the *limit* (in terms of memory occupancy) is about the L1 capacity (32KB in KNL), but it is more closed to the L2-cache (1MB in KNL). It because of this, that we consider as *limit* the capacity of L2-cache instead of L1-cache for KNL. Similarly to *taskloop*, in *grouping-op* the *limit* (*#operations*) is set as the total number of FLOPS divided by the number of cores available. As the number of cores in KNL (64) is higher than the number of cores in our *NUMA*-based platform (16), the limit is considerably smaller in KNL than in Xeon. When Xeon is used the larger are the matrices, the larger is the *limit*, which, as previously commented, is necessary to achieve a good performance using this approach when the data is interleaved. However, on KNL, there are not substantial differences if the data is interleaved or not, obtaining a good performance when the *limit* is set as the total number of FLOPS divided by the number of cores available on both cases.

Once the parameters have been tuned, we perform a number of experiments in order to clarify what benefit might be gained from adopting the *off-line auto-tuning*.

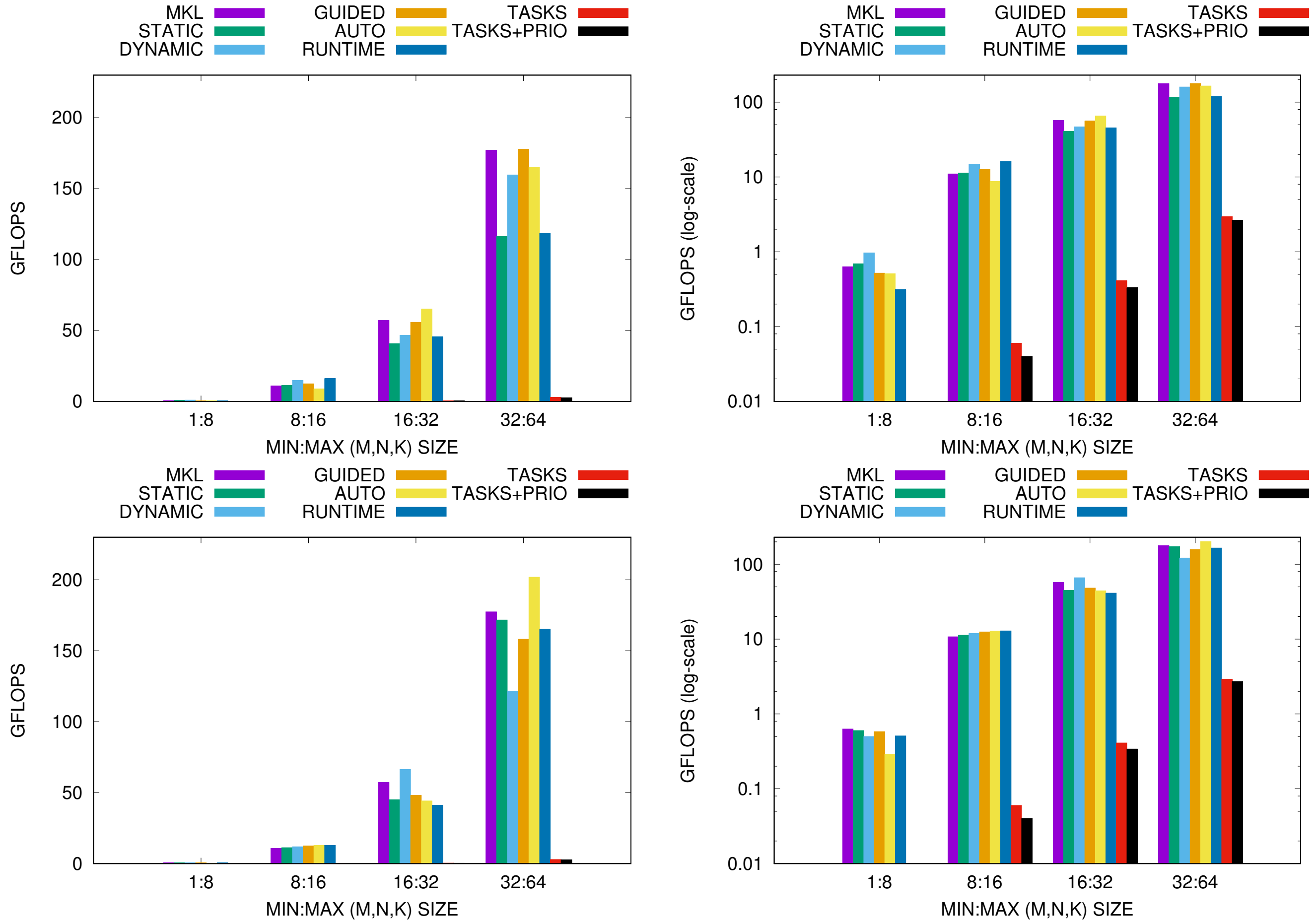

Figure 2. GFLOPS achieved by *MKL*, *parallel for*, *task* and *task+prio* to compute 10,000 *DGEMMs* for different size ranges (MIN:MAX): (1:8), (8:16), (16:32) and (32:64), interleaving (top) and not interleaving (bottom) the data between *NUMA* nodes.

Figure 6 and 7 graphically illustrates the performance achieved by each of the tunable approaches *taskloop*, *grouping-op* and *grouping-mem* on both architectures, Xeon and KNL, respectively. We show the performance achieved by each of the approaches, *taskloop* (TL), *grouping-mem* (GM) and *grouping-op* (GO) in Fig. 6 and 7, using *off-line auto-tuning* compared with the peak performance achieved in the cases evaluated previously (Fig. 3, 4 and 7), as well as with the performance achieved by the best *OpenMP parallel for* schedulers (Fig. 1 and Fig. 2), *guided* and *dynamic* when data is interleaved and non-interleaved, respectively. We also include the results obtained using the *MKL* routine, *cblas_dgemm_batch*.

First, we focus on the NUMA-architecture (Xeon). As shown, depending on how the data is stored in memory, we can see a different trend. For instance, *grouping-op* is able to obtain a good performance using *off-line auto-tuning* when data is interleaved. However, when data is non-interleaved, the tuning used turns to be inefficient, causing an important fall in performance. For *grouping-op* a different tuning must be used depending on how data is mapped. When data is non-interleaved, a big *limit* makes that those tasks that have to access to the memory of the other NUMA nodes consume much more time than those tasks which do not have to, causing an important unbalancing. We can see a similar effect in *taskloop* (in Fig. 3), *MKL* and *parallel for static* (in Fig. 1).

On Xeon, the approach based on *taskloop* is not able to deal with different data mappings, suffering an important fall in performance when data is non-interleaved, like in *grouping-op*. Unlike the previous approaches, *grouping-mem* is able to deal with different memory mappings, achieving a similar performance independently on how data is mapped in memory. In particular, this approach is proven to be faster than *parallel for dynamic* for non-interleaved data mappings and slightly slower than *parallel for guided* for interleaved data mappings. The efficiency of this approach is found in the way that the set of *DGEMMs* are assigned to tasks. Also, using as *limit* the size of L1-cache, we guarantee an efficient use of the on-chip memory hierarchy. Basically, this

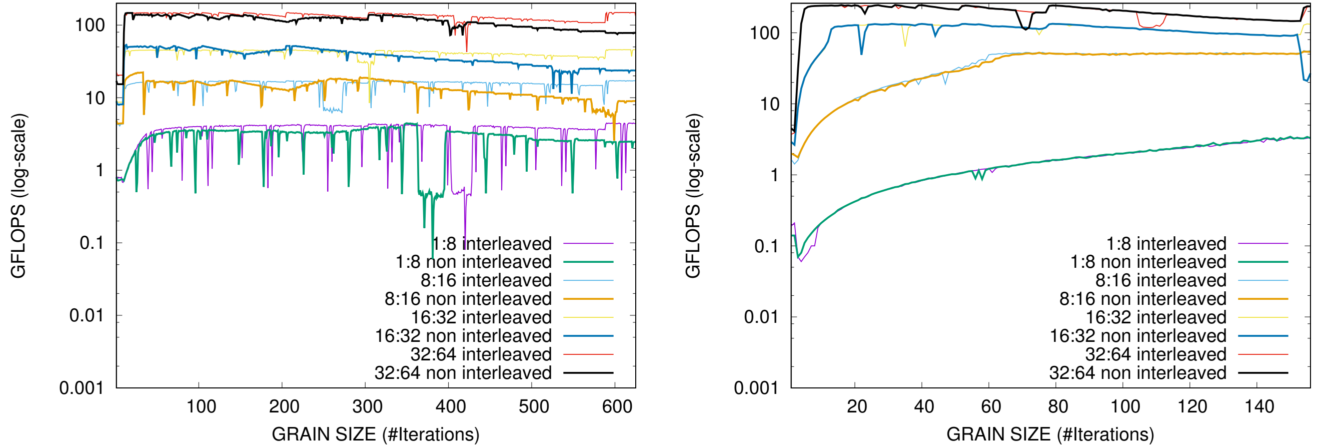

Figure 3. GFLOPS achieved on Xeon (left) and Xeon KNL (right) by *taskloop* increasing the *grainsize* from 1 until 10,000/#cores to compute 10,000 *DGEMMs* for different size ranges (MIN:MAX): (1:8), (8:16), (16:32) and (32:64).

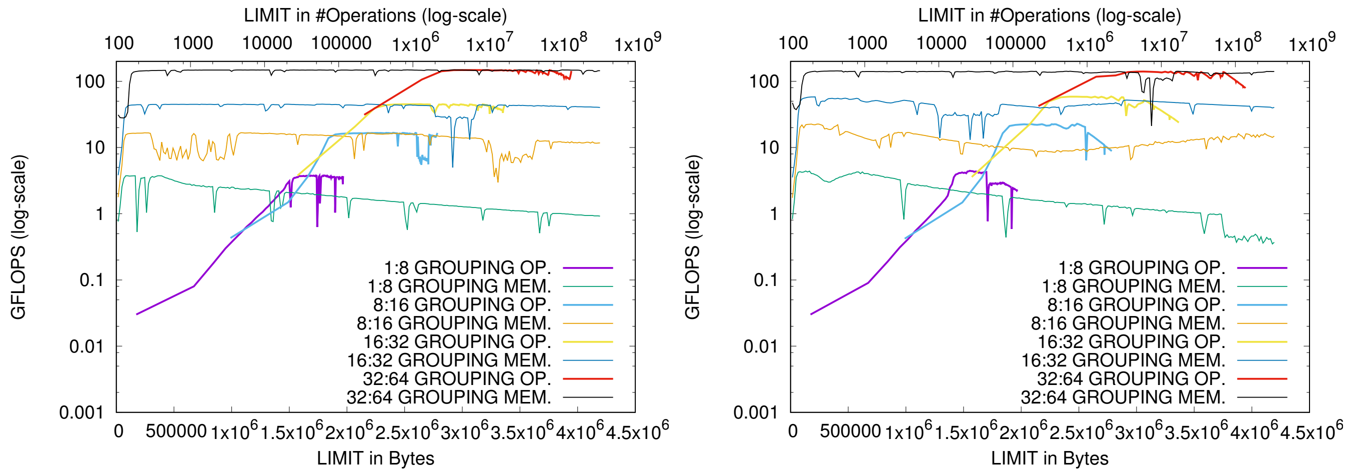

Figure 4. GFLOPS achieved on Xeon by *grouping* increasing the memory capacity from 16KB until 4MB and increasing the #operations from TotalFlops/10,000 until TotalFlops/#cores to compute 10,000 *DGEMMs* for different size ranges (MIN:MAX): (1:8), (8:16), (16:32) and (32:64), for interleaved data (left) and non-interleaved data (right).

approach is able to deal with *NUMA* architectures in the way that those tasks that have to access to memory of other nodes (consuming more time) can be overlapped with those tasks that do not have to. In this case, we have a higher number of tasks executed on those nodes that have more *DGEMMs* stored in memory.

It is important to highlight that the *grouping*-based approaches are able to outperform the *MKL* batch routine on Xeon. The benefit of *grouping* is even bigger when the data is non-interleaved (Fig. 6).

Now, we focus on KNL. Unlike Xeon, *grouping-op* achieves a better performance on KNL. In particular, we do not see an important fall in performance when using auto-tuning on this approach independently is the data is interleaved or not. In fact, *grouping-op* is able to outperform *grouping-mem*. This is mainly because of the *limit* used on this architecture for *grouping-mem*. As commented, we used the L2-cache capacity as *limit*, however, the peak performance is achieved when the *limit* is bigger than L2-cache (see Fig. 5). This approach is not as effective as in Xeon being more difficult to tune in KNL. Although using *taskloop* is achieved a high GFLOPs number, this approach is also difficult to tune. As we can see in Xeon, we find an important fall in performance when using auto-tuning on KNL for *taskloop* as well. The differences among the throughput obtained by *MKL* batch and by the auto-tuned approaches are not so big as in Xeon.

F. Overview

The proposed approaches, *grouping-op* and *grouping-mem*, have proven to be an efficient way to deal with *batch variable*. They are able to outperform the *MKL*

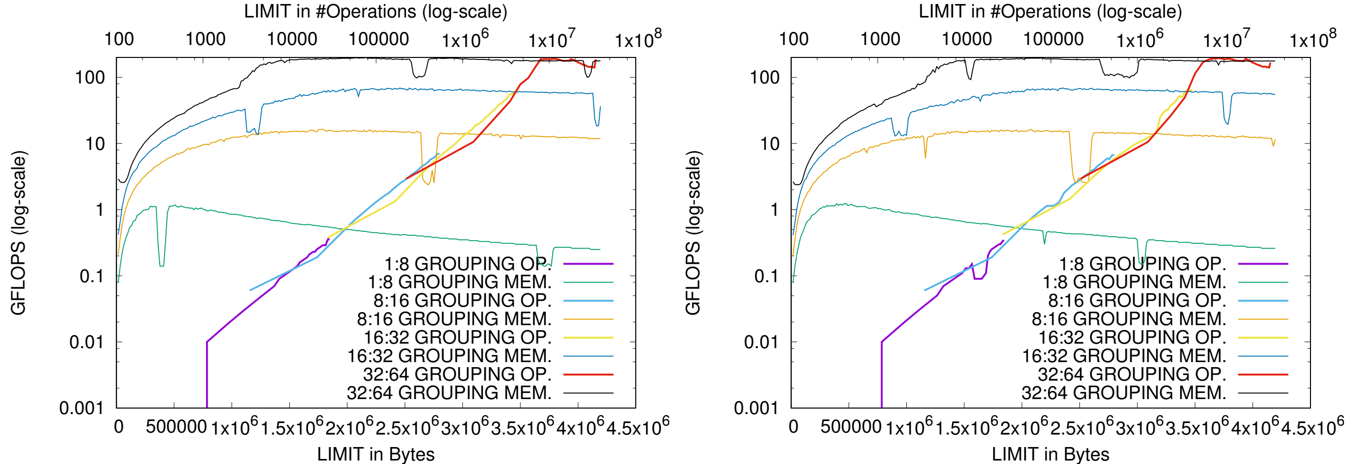

Figure 5. GFLOPS achieved on KNL by *grouping* increasing the memory capacity from 16KB until 4MB and increasing the #operations from TotalFlops/10,000 until TotalFlops/#cores to compute 10,000 *DGEMMs* for different size ranges (MIN:MAX): (1:8), (8:16), (16:32) and (32:64), for interleaved data (left) and non-interleaved data (right).

cblas_dgemm_batch routine. The benefit achieved is different depending on the size of the matrices and the platform used. The benefit of using these approaches decreases, increasing the size of the matrices. When one *DGEMM* is big enough, i.e. the memory occupancy of C = alpha × op(A) × op(B) + beta × C is at least as big as L1-cache, the computation of just one *DGEMM* can saturate the computational capacity of one core. On Xeon (Fig. 6), the use of *grouping-mem* has shown better results than *grouping-mem* being 3.31× and 1.65× faster (in terms of GFLOPS) than *MKL* batch for the ranges (1:8) and (32:64) respectively for non-interleaved data, and 5.75× and 0.92× faster for interleaved data. When auto-tuning is considered the performance changes, being the *grouping-mem* with a *limit* equal to L1-cache (see Table II) 3.44× and 1.88× faster than *MKL* batch for the same ranges for non-interleaved data and 6.38× and 0.84× faster for interleaved data.

As in Xeon, in KNL the *grouping-mem* is proven to be better than *grouping-op* (Fig. 7)). We achieve an extra throughput about 1.97× and 1.14× (in terms of GFLOPS) than *MKL* batch for the ranges (1:8) and (32:64) respectively for non-interleaved data, and 1.87× and 1.14× faster for interleaved data. However, when using auto-tuning the *grouping-op* turns to be faster than *grouping-mem*. Unfortunately, the use of the auto-tuning strategies proposed in this work are not as effective as in Xeon, being the *MKL* batch routine faster.

The *parallel for OpenMP pragmas* present a good performance, in particular the *guided* and the *dynamic*, to deal with *batch variable* on Xeon architectures, except for very small matrices (see Fig. 6). However, on KNL, the performance of the *parallel for pragmas* are smaller with respect to the other approaches tested (see Fig. 6-MIC).

IV. Conclusions

We have compared several strategies to deal with *batch variable*. Although, the use of *parallel for dynamic* has been used in other works to deal with this kind of problem, we have proven that there are other interesting strategies that can outperform the use of *parallel for dynamic*. Also we have analyze the performance of the *MKL cblas_dgemm_batch* routine. We have proposed one strategy called *grouping*, which can efficiently deal with the differences among the independent works of the *batch*, in particular for small matrices. We have proven that the way in which the data is stored in memory have important consequences in performance on the *NUMA* architecture. In this sense, the only approach that is able to deal with different kind of data storage is the one proposed, being the most effective the one being the that based on memory occupancy (*grouping-mem*). However, these approaches must be tuned to guarantee a good performance. We have explored two different auto-tuning strategies, the most effective one being the that based on memory occupancy (*grouping-mem.*) for Intel Xeon and that based on number of operation (*grouping-op*) for Intel Xeon PHI.

Given the results shown in this work, we want to analyze other BLAS routines, as well as sparse operations, such as SpMV, and the efficiency of these optimizations on real applications. Also, as the effectiveness of the auto-tuning strategies presented in this work are, in some cases, far from the peak performance achieved, in particular on KNL, we want to explore other approaches based on on-line auto-tuning instead of off-line auto-tuning.

Acknowledgement

This project has received funding from the European Union’s Horizon 2020 research and innovation programme

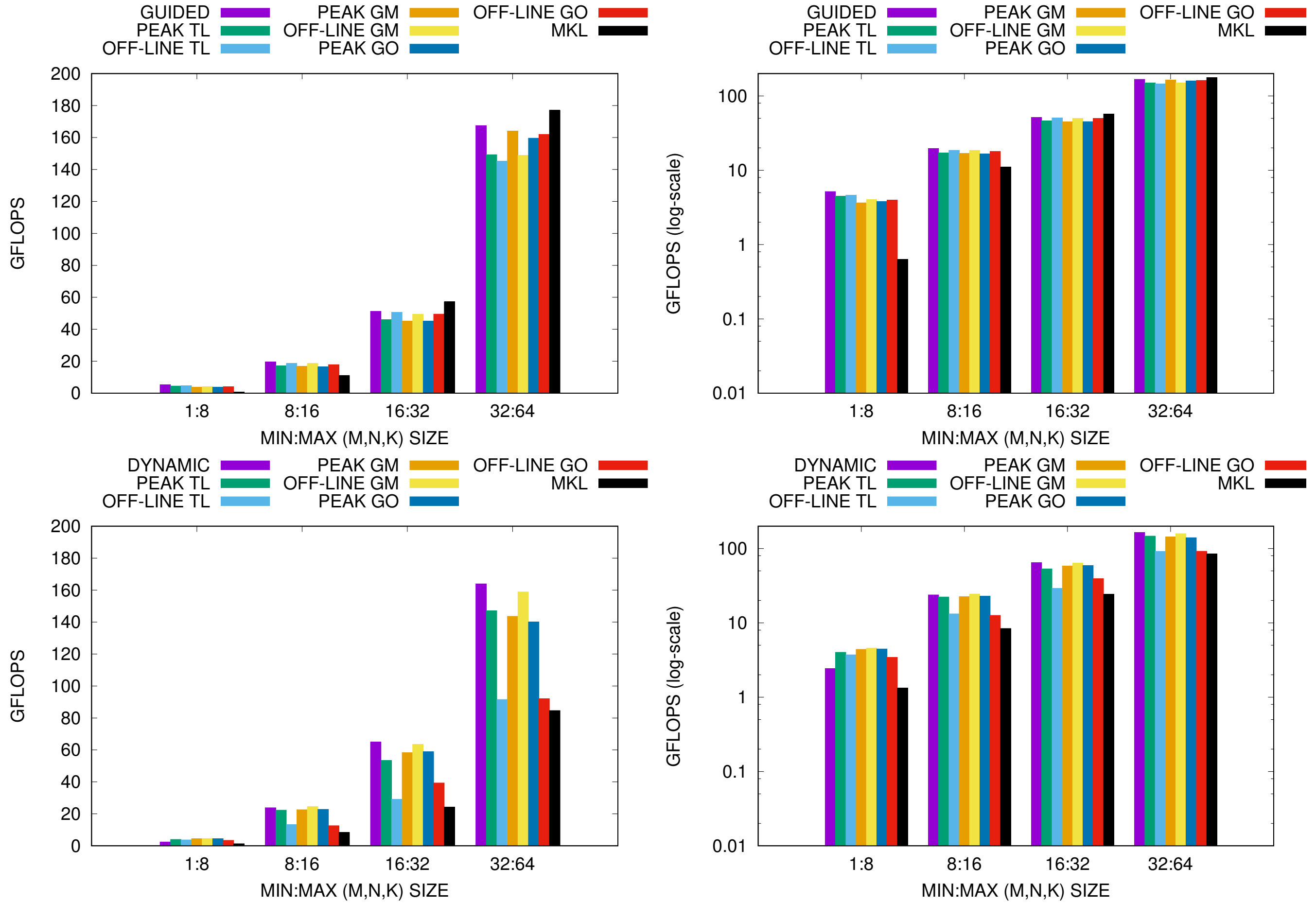

Figure 6. GFLOPS achieved on Xeon by each of the tunable approaches (*grouping-mem*, *grouping-op*. and *taskloop*) to compute 10,000 *DGEMMs* for different size ranges (MIN:MAX): (1:8), (8:16), (16:32) and (32:64), when the data is interleaved between the *NUMA* nodes (top) and the data is non-interleaved (bottom).

under grant agreement No 720270 (HBP SGA1), from the Spanish Ministry of Economy and Competitiveness under the project Computación de Altas Prestaciones VII (TIN2015-65316-P) and the Departament d'Innovació, Universitats i Empresa de la Generalitat de Catalunya, under project MPEXPAR: Models de Programació i Entorns d'Execució Paral·lels (2014-SGR-1051).

REFERENCES

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects," *Journal of Physics: Conference Series*, vol. 180, no. 1, 2009.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[3] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau *et al.*, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016. [Online]. Available: http://arxiv.org/abs/1605.02688

[4] A. Abdelfattah, M. Baboulin, V. Dobrev, J. J. Dongarra, C. W. Earl, J. Falcou, A. Haidar, I. Karlin, T. V. Kolev, I. Masliah, and S. Tomov, "High-performance tensor contractions for gpus," in *International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA*, 2016, pp. 108–118.

[5] W. Hackbusch, "A sparse matrix arithmetic based on h-matrices. part i: Introduction to h-matrices," *Computing*, vol. 62, no. 2, pp. 89–108, 1999.

[6] S. D. Relton, P. Valero-Lara, and M. Zounon, "A comparison of potential interfaces for batched BLAS computations," Manchester Institute for Mathematical Sciences, The University of Manchester, UK, MIMS EPrint 2016.42, 2016.

[7] A. Charara, D. E. Keyes, and H. Ltaief, "Batched triangular dense linear algebra kernels for very small matrix sizes on gpus," *ACM Transactions on Mathematical Software*, 2017.

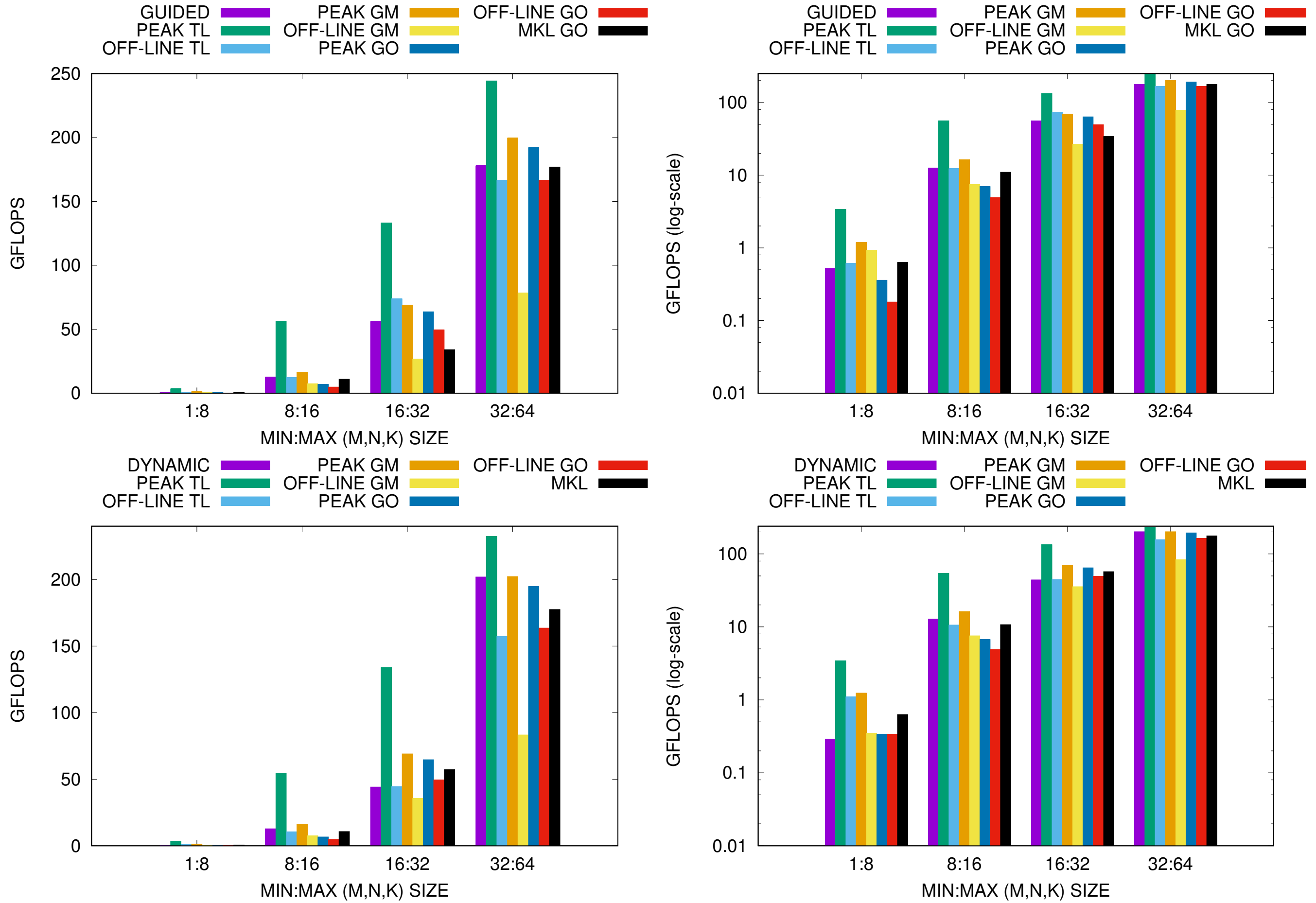

Figure 7. GFLOPS achieved on KNL by each of the tunable approaches (*grouping-mem*, *grouping-op*. and *taskloop*) to compute 10,000 *DGEMMs* for different size ranges (MIN:MAX): (1:8), (8:16), (16:32) and (32:64), when the data is interleaved between the *NUMA* nodes (top) and the data is non-interleaved (bottom).

[8] A. Abdelfattah, A. Haidar, S. Tomov, and J. J. Dongarra, "Performance, design, and autotuning of batched GEMM for gpus," in *High Performance Computing - 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*, 2016, pp. 21–38.

[9] P. Valero-Lara, P. Nookala, F. L. Pelayo, J. Jansson, S. Dimitropoulos, and I. Raicu, "Many-task computing on many-core architectures," *Scalable Computing: Practice and Experience*, vol. 17, no. 1, pp. 32–46, 2016. [Online]. Available: http://www.scpe.org/index.php/scpe/article/view/1148

[10] 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.

[11] P. Valero-Lara, I. Martínez-Perez, 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.

[12] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "On the development of variable size batched computation for heterogeneous parallel architectures," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016*, 2016, pp. 1249–1258.

[13] R. C. Whaley and A. M. Castaldo, "Achieving accurate and context-sensitive timing for code optimization," *Softw. Pract. Exper.*, vol. 38, no. 15, pp. 1621–1642, 2008.

[14] A. Sodani, "Knights landing (knl): 2nd generation intel xeon phi processor," Intel Corp., Tech. Rep., 2017.

[15] I. Masliah, "Evaluating the impact of intel knl memory settings on performance through case studies," Inria, Tech. Rep., 2017.

[16] J. Dongarra, I. Duff, M. Gates, A. Haidar, S. Hammarling, N. J. Higham, J. Hogg, P. Valero-Lara, S. D. Relton, S. Tomov, and M. Zounon, "A proposed API for batched basic linear algebra subprograms," The University of Manchester, UK, MIMS EPrint 2016.25, 2016.